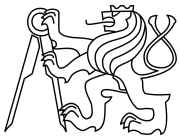
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Constraint Models for Planning and Scheduling
Student:	Bc. Martin Procházka
Supervisor:	prof. RNDr. Roman Barták, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Student will first familiarize with constraint satisfaction techniques for solving planning and scheduling (P&S) problems, in particular with the design of constraint models for P&S problems. Based on this study, the student will propose a constraint model for a specific problem, for example, from the area of space applications. Then he will implement the model for a selected constraint solver, such as SICStus Prolog, and finally he will experimentally evaluate it.

References

Roman Barták, Miguel A. Salido, Francesca Rossi: New Trends on Constraint Satisfaction, Planning, and Scheduling: A Survey. The Knowledge Engineering Review, Vol. 25:3, 249-279. Cambridge University Press, 2010.
Philippe Baptiste, Claude Le Pape, Wim Nuijten: Constraint-Based Scheduling, Applying Constraint Programming to Scheduling Problems. Springer, 2001.
Rina Dechter: Constraint Processing, Morgan Kaufmann, 2003.
Martin Kolombo, Roman Barták. A Constraint-based Planner for Mars Express Orbiter. In Proceedings of MICAI 2014, Part II, LNAI 8857, pp. 451- 463, Springer, 2014.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 3, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Constraint Models for Planning and Scheduling

Bc. Martin Procházka

Supervisor: prof. RNDr. Roman Barták, Ph.D.

9th May 2017

Acknowledgements

I thank my father for taking his time to correct the grammar of this thesis. Also I thank my supervisor for pointing me the right way, so I could find a topic that was interesting and challening. Last but not least I thank my family for their patience.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 9th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Procházka, Martin. *Constraint Models for Planning and Scheduling.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Cílem této práce je navrhnout výpočetní model pomocí technologie constraint programming, který by byl schopen popsat a vyřešit konkrétní plánovací problém. Problém je definován pomocí plánovacího jazyka PDDL a je zaměřen na offshore větrné farmy, jejich údržbu a plánování zdrojů, které jsou s údržbou spojeny. Model který navrhujeme je založen na výzkumu Graphplan algoritmu a jeho využití v paradigmatu constraint programmingu pro plánování. Popisujeme všechny navržené akce a jejich případnou transformaci z původní PDDL domény do logického modelu constraint programmingu. Nakonec provádíme analýzu výsledků a navrhujeme vylepšení nebo návrhy na jiné implementace modelu pro budoucí práci.

Klíčová slova plánování, constraint programming, rozvrhování, PDDL, plánovací graf, programování pomocí omezujících podmínek

Abstract

The purpose of this thesis is to propose a CSP model that would be able to describe and solve a specific planning problem. The problem is defined by a PDDL domain and it is related to an offshore wind farm maintenance and to the planning of the resources required. The model we propose is based on the

research of the Graphplan algorithm and its use in the constraint programming approach to planning. We describe all the actions and their eventual mapping from the original PDDL domain to the logical model of the constraint programming. Finally we analyze the results and propose improvements or ideas on a different implementation of the model for future work.

Keywords planning, constraint programming, CP, PDDL, scheduling, durative actions, integer domains, planning graph

Contents

List of Figures

List of Tables

Introduction

Today's society is very developed because of the excessive use of energy which we use to power our factories, hospitals and cars. In the past few years there has been a huge shift in focus to renewable sources of energy as an alternative to the traditional sources such as coal, oil and nuclear power. One of the most accessible sources for certain areas are wind turbines, which generate the clean energy with relatively low costs, which have become even lower in the past few years. [4].

The problem with the wind turbines is that they possess some unwanted properties such as noise, which prevents them from being used on a larger scale onshore or close to the cities. This can be solved by constructing them further away from population centres or, if there is a lack of space, offshore. The European Union is the biggest user of these offshore facilities with UK being responsible for 53% of the worldwide production. [5]

Offshore wind turbine facilities are however very costly to maintain although the life expectancy of a wind turbine is around 20 years. While the repairs carried out onshore can cost thousands of euros, repairs carried out offshore can cost millions of euros [1]. The reasons for this are more constraints and logistic costs, as a blade or gearbox replacement require huge transportation capabilities. In addition the weather constraints, when certain means of transport can't operate in high winds and/or high waves. The planning that seeks to minimise these costs is therefore of the utmost importance.

Chapter 1

Light problem introduction

As was outlined in the introduction, the problem is in the domain of wind turbine maintenance planning. For the investigation of the current solution a broad non-technical definition is satisfactory. Detailed decomposition will be presented while presenting the newly made solution.

We are presented a real world situation, where there are wind turbine farms offshore, which consist of several wind turbines and a basic infrastructure for transportation of the people providing maintenance between those turbines (meaning they don't have to use any special means of transport to get from one turbine to another). Also they include a port for ships and barges. The port is the only access point that can be used for more complicated repairs where heavy parts are needed.

The wind farms can also be reached by a helicopter, which can transport engineers to site for minor repairs e.g. a manual restart of a turbine. A helicopter however faces a huge limitation of fuel as it can't land on the wind farm and has to get back to the nearest heliport before its reserves run dry. For more complicated or numerous operations the helicopter can't hover and wait for the engineers to finish their work, but rather has to serve as a pick up and drop off service.

There are different land ports which contain boats for rent and heliports that contain helicopters for rent. From these points the engineers may be loaded along with parts needed for the repair.

The parts are usually located in warehouses or can be manufactured in factories if necessary, but this however takes a long time. The parts can be delivered to ports by cars, which are not a part of the planning abstraction (they can be always delivered at a certain speed by some rented car). All land locations are interconnected by land links.

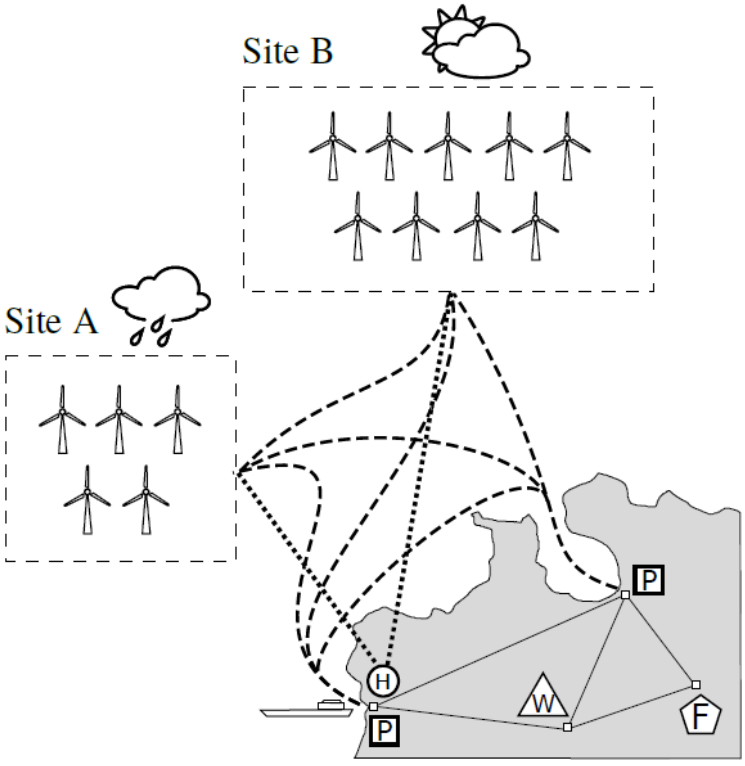


Figure 1.1: An image of the domain as presented in [1]

Chapter 2

Baseline solution

The problem that is researched in this work is defined in the paper [1], but is solved by a different technique, which we will try to improve on. To understand the differences in the solution methods and why the one proposed is better suited for this task, the former method has also been researched and will be described in the section

We have investigated the current solution proposed in the paper to understand the methods used to model and solve the problem. There are some approximations and logical omissions regarding the realness of the model and the quality of the solution.

2.0.1 Prior solution and considerations

The solution that was available before the automated planning solution was implemented (although it is probably still used even now) was planning by hand. When a maintenance operation was required, the plan was designed by a logistics expert. The objective of this expert was the same as ours - to minimise the total cost associated with the operation while also minimising the downtime of the failed turbines. [1]

The components that can break also differ in their size and so does the complexity of the repair. Some repairs like a manual reset are quick and don't require any additional components while a blade replacement can take weeks to complete and requires specific components (blades) to be transported to site and replaced. Companies thus associate the maintenance actions with fault categories and produce generic plans which can be applied as needed. Those plans are not yet grounded and the grounding procedure is dependent on the problem at hand.

The difficulties and constraints that an expert has to take into consideration while grounding the plan were mentioned in the introduction.

An important note is that the plans and the domains are static. Although the faults happen dynamically and new problems may arise during the plan

execution, it is not a part of this work as it would increase complexity. 'Dynamic' plan estimation may be done by simple restart of the planning process with updated initial state newly including new faults and updated positions of the personnel (new goal is not necessary). Because the main costs are usually associated with the repairs and not with the downtime, it is usually more cost efficient to wait with the repair until regular revision tasks are scheduled and then do all of the tasks in one plan.

Taking into consideration that the plans are static it is not a big surprise that the problem is also fully observable. New problems can't arise on site, although in reality they usually do. This might be a thing to consider for the task - to make a buffer time which could cover some basic operations on the platform (e.g. one dummy restart per turbine inspected).

2.1 Problem encoding

Planning problem in the former solution is encoded by the Planning Domain Definition Language (PDDL 2.2 language) and solved by the Optics solver. The problem has been encoded in two encodings - WINDY-SIMPLE and WINDY-COMPLEX. The paper states that solutions have been found only for the simplified domain.

The simplified domain is used to generate a general plan without cost optimization. This is good for the goal of generating a suitable plan, but it fails to cover the main objective - to minimise cost.

This is covered by the WINDY-COMPLEX domain where there is implemented the required concurrency (RC) constraint from PDDL 2.2. The RC used was encoded such that the *lease* actions could overlap and would cover all the actions required as child actions. For example when a helicopter flies, drops off engineers to perform operations on the turbines and then collects them and returns, it has to be leased all the time it performs those operations.

However finding an optimal plan or just a solution with regards to the optimisation criterion has proven impossible for all but the most trivial cases. The cost estimation had to be moved to the post processing stage of the solution and thus a lot of the information for the search is lost and the Optic solver can't make well informed decisions during the search and optimise efficiently.

2.1.1 Note on the reproduction of the results

The objective function results were not actually published in the paper nor was there a detailed description of the problem. There were just types of repairs, with their respective counts required for each instance and the times it took the solver to solve these instances. This wasn't a good baseline for us to be able to compare our results, but thanks to Dr. Pattison's generosity we

received the full pddl files along with the modified version of the Optics solver they used.

Chapter 3

Technologies used in the baseline solution

As was mentioned above, the paper used the PDDL 2.2 encoding and Optics solver. In this section we would like to analyse both to see what advantages or disadvantages this model presents and if it can help us in the construction of the constraint programming model.

3.1 PDDL

PDDL or Planning Domain Definition Language is a language that is an attempt to create a common formalism for the planning tasks and its use has been implemented into a number of solvers. In other words it is a standard encoding language for planning tasks.

The original language has been developed for the problem specification of the AIPS-98 planning competition. It was one of the means that could reinforce the empirical evaluation of the solutions by forcing all of the competitors to encode the problem in one common code providing a reasonable constraint that unified the problem definition [6] (otherwise the different encoding of the problems could lead to misinterpretation of the problems).

The language supports the following features - basic STRIPS style actions, conditional effects, universal quantification over dynamic universes (object creation and destruction), domain axioms over stratified theories, specification of safety constraints, specification of hierarchical actions composed of subactions and subgoals and management of multiple problems in multiple domains using differing subsets of language features so different solvers with different features implemented could share domains.

PDDL serves as a prescription of the dynamic behaviour of a domain. There are objects modified by actions. For each state (a collection of objects) we can form predicates, which are basically questions about the state with

```
(:types         turbine - locatable
                turbine port - landable
                airport seaport - port
                seaport airport warehouse ... turbine  landable - location
                     ...
)

(:predicates
               (operating ?t - turbine)
               (at ?loc - location ?obj - locatable)
               (link-sea ?from - location ?to - location)
               ...
)

(:action disable-turbine
                :parameters (?t - turbine)
                :precondition (operating ?t)
                :effect (and (not (operating ?t)) (not-operating ?t))
)
```

Figure 3.1: Basic PDDL definition of the modelled problem

yes/no answers (e.g. a predicate *isInCar(x)* which is true when x is indeed in the car). Actions transform the world from state A to state B while modifying the affected state B as prescribed. To finish this brief enumeration, there are also initial and goal states defined. The work of the planner is to find a sequence of actions that transforms the world from the initial state to the goal state.

The aforementioned features could also fit on the STRIPS problem definition, but it lacks certain levels of expressiveness which result in some problems being unmodellable.

One of the great features of PDDL is an object hierarchy. In the domain definitions we can provide a hierarchical graph – possibly a tree as the descendancy is defined as an ISA relation. Then we can define actions that use this hierarchy and thus the effects and preconditions can be defined for all objects of the specified type and its subtypes.

3.1.1 Example from the PDDL definition of the problem

Here we present an excerpt of the WINDY-SIMPLE problem definition to demonstrate the PDDL features.

On the figure 3.1 we can see the hierarchy of the objects. Provided by

the hierarchy we can see basic predicates available for a specific object – the turbine. Since the turbine is not extended further, it is a very simple predicate indicating only whether the turbine is in operation.

More interesting is the predicate *at* , which indicates if a locatable appears on a location. Locatable and a location include a lot of different objects and this predicate is universal for all of them.

The action is a simple one, possibly representable by a simple STRIPS representation. We can see that at any time we can disable the turbine if it was (precondition) operating before and the effect is that it is not operating anymore.

3.2 PDDL2.2

The PDDL2.2 language [7] is derived from the PDDL2.1 language developed by Fox and Long [8]. The PDDL2.2 bases on the basic PDDL language (a valid model of PDDL2.1 is a valid model of PDDL2.2 and a valid PDDL model is also a valid PDDL2.1 model) while adding certain features that are missing from the former and which allow us to model and solve an extended class of problems. The PDDL2.2 features has been extensively used in our base solution to which we compare our solution, so we'll try to pinpoint features that are used and provide examples.

The language has been developed for the *4th International Planning Competition* (IPC-4). From the PDDL2.1 it takes all its three levels of expressive power and adds on top of it derived predicates and timed initial literals. For our researched problem definition, the metrics and the durative actions from the PDDL2.1 are the most important.

One other important feature the PDDL2.1 introduced was a concurrency of actions. Until then the plans tended to be sequential, but since the durative actions were introduced, time had to be taken into account. With concurrency there were introduced questions about the constraints between the actions as actions can't sometimes logically take effect at the same time (e.g. an action moving a robot to the left can't run at the same time as an action moving the robot to the right). Those questions are tackled in the PDDL2.1 specification [8] and they very much resemble the constraints that are present in the Graphplan algorithm. [9]

3.2.1 New features of PDDL2.{1,2}

From PDDL2.1 the metrics have been used. Those *metrics* are used for the specification of the object function. This can provide a measure for the planner to search for the best plan according to the specified optimisation criterion. The language does not state however whether the solver will use the metric efficiently during the search process (to prune the branches) or whether the estimation will happen post hoc after a valid plan is found.

Also from PDDL2.1 there is a very important definition of *durative actions*. The durative actions are in their core still actions which rely on the logical changes caused by the application of the action. On top of that however they require certain conditions to be satisfied during the whole run of the application of the action. Those are called *temporarily annotated* conditions and effects. The annotation of the conditions makes it possible for us to differentiate whether the conditions must hold on the start of the action, on the end or over the interval from the start to the end with the endpoints excluded. The exclusion is necessary to be able to model situations, where another action can happen exactly after the end of the durative action. For example consider a hovering helicopter where an engineer is climbing on-board. The helicopter may begin its journey back just as the boarding procedure completes.

A new feature from PDDL2.2 that is used in the previous work are *timed initial literals* (TILs). Those are basically time windows during which facts either do or don't hold. They are known in advance and thus are deterministic external and unconditional events.[7] An easy example of TILs is a daylight - we know always in advance the timing of light and dark hours.

3.2.2 Example

As we can see on the code on the figure 3.2 the features we listed are utilised in the definition of the domain and also in the problem definition. Very much utilised are the *durative action* features. As you can see, there is a duration defined for the action. Also there is a condition over all, that the helicopter must be leased and in the port to be refuelled. At the end of the action the remaining time is updated to its maximal value.

The other two features are used in the problem definition. A *metric* is defined as a total lease cost while the objective is to minimise it.

As an example for the TILs there is the example already mentioned, the TILs depicting the light hours of the day.

3.3 The OPTIC solver

The PDDL language is very useful in the means of definition of the problem domain and the concrete problem. It is however the solvers which do the hard work of interpretation of the PDDL language and production of valid and as much as possible optimal solutions.

The solver that has been used by the authors of the reference paper was an OPTIC solver [10]. It is fitted to interpret PDDL3 language and implements a large portfolio of PDDL3 features like preferences and preference constraints. The constraints are encoded using dummy steps and, without getting into much detail, provide us with modelling options of predicate relations like *sometimes-before* (a car can go to a city B only if it has visited

```
(:durative-action refuel-helicopter
                :parameters (?h - helicopter ?port - airport)
                :duration (= ?duration 0.5)
                :condition  (and  (over all (leased ?h))
                                  (over all (at ?port ?h)))
                :effect
                (and (at end (assign (range-remaining-time ?h)
                                     (range-max-time ?h))))
)

;PDDL problem specification
(:metric minimize (total-lease-cost))

(at 14 (not (daylight)))
(at 24 (daylight))
...
```

Figure 3.2: A domain definition and a problem definition excerpt in PDDL2.2

a city *A* sometime before) and *sometimes-after*. Those features (and many more) are however not used in the researched solution.

3.3.1 The POPF solver

The OPTIC solver is strongly based on the Coles' previous solver - the POPF solver[11]. Most of the features used are already implemented in POPF and Optic solver provides extensions which are not used to model the problem. The POPF solver officially supports only PDDL2.1, so it would seem that the TILs must be a part of the OPTIC extension. This is however not the case as the TILs are already present in the POPF solver and are actually used as an implementation of a preference *hold-during*.

The POPF solver is again based on a solver Colin and serves as an extension to it. Colin is a solver that produces partially ordered plans. Partial ordering provides many advantages - the plans that are found are not grounded and the grounding procedure may wait until further constraints arise.This is very useful in the context of the process that is currently used to solve our domain problem as was noted in the section 2.0.1. However the optimal grounding of a plan is NP-hard and the construction of partial-order plans has proven difficult.

POPF thus tries to exploit the advantages of partial-order planning but at the same time uses a new paradigm in the plan construction - the forward chaining. Forward chaining or forward state-space search is a search where we start with the starting propositions and facts and try to find the goal

through the depth first search. The nodes in the search tree in Colin are states describing the partial plan with constraints, steps, facts that hold in the state and records of temporal constraints that are in effect.

The strength of the FC is that there is no need to search and resolve threats in the partial plan as it imposes a total order on actions. The total ordering prevents any threats to the previously added preconditions and effects from the actions newly added as they come after all the actions already in. Also, because of the temporal nature of the problem, the newly added actions must not threaten any invariants of ongoing durative actions.

The main weakness of FC comes from the early commitment it implies and the lack of parallelism of non-durative actions. The early commitment is dangerous as it determines the ordering of the actions very early even when it wouldn't be necessary. Trying to correct a plan that has taken an action that it shouldn't have in the starting steps means a regeneration of the whole tree with the correct order. The authors however state, that the tree generation and backtracking is very rapid. [11]

In the temporal case the planning considers both starts and ends of actions which makes, along with total ordering, the search very difficult. As an example taken from [11]: consider the example of two durative actions, A and B, with start points $A_{\vdash}$ and $B_{\vdash}$ and ends $A_{\dashv}$ and $B_{\dashv}$. Suppose that B is longer than A, and the actions do not interfere with one another at the start, but due to their end conditions $B_{\dashv}$ must precend $A_{\dashv}$. If the planner chooses to order the starting actions in order $A_{\vdash} \ldots B_{\vdash} \ldots B_{\dashv}$, the temporal constraints will turn out to be unsatisfiable and require the planner to backtrack through all the intermediate decisions until it orders $B_{\vdash}$ before $A_{\vdash}$.

Another problem of the FC are deadlines that arise during the search as the point of failure is at the deadline itself. The failure of the deadline is however not an instantaneous problem and the reasons for the failure may have emerged on the very start of the plan construction. The effect is the same as with the previously stated weaknesses of FC - excessive backtracking.

The POPF planner however proposes and implements additional data structures and algorithm modifications which allow to support partial order planning with FC. Basically it newly records achievers and deleters (steps with actions that add or delete a fact) of the required facts and promotes them to be placed before/after steps which require them. As an interesting extension the POPF planner also contains similar functionality for the numeric temporal planning. Steps are however recorded if they have an effect on a value of a numeric variable v, either instantaneous or continuous, or if they have some dependency on v. The part of algorithm to use this feature is very similar to the one with facts.

Finally, after finding the partially-ordered plan which means that the constraints over the actions are chosen, the plan needs to be grounded and checked for consistency. Until now the solver has created a solution with uplifted constraints and although it has chosen the ordering of the actions, it is yet to

be found out if they are truly capable of forming a totally-ordered plan. The planner also offers a heuristic based on calculation of minimal timestamps that can be assigned to the steps according to the facts and/or numeric values required.

3.3.2 Solver(s) summary

We can see that the OPTIC solver is very sophisticated and that its solving mechanism has been very much researched and optimised. It was demonstrated on the description of POPF solver - the core of OPTIC which contains most of the features that has been actually used. The algorithm type that is utilised to find a solution is of forward chaining type and is basically a highly developed search with its interpretation of constraints. On one hand this is very different from the CP paradigm that is based mostly on declaration, but on the other hand the planning domain problem solutions are in a way alike so we might use the knowledge we have gained from the research in our solution.

Chapter 4

Constraint programming

Constraint programming (CP) is a paradigm for solving combinatorial search problems. To understand what it really means, we must define both terms used in its name.

Constraints are logical relations between two objects. They don't have to be anyhow bound to some computational area for us to understand what it represents. If we say that a dot is in a square, we have provided a relation between two objects which constraints the reality in a way, that this statement holds. Of course we have to be absolutely sure that it is indeed the case, as we have put a *constraint* on the observed world by stating that it is true. It is obvious that we have narrowed down the number of possible observed worlds (although from infinity to infinity since we didn't bound our system in any other way) and that we can continue to make new statements which should ultimately define the observed system enough so we would be able to extract some interesting information from within.

This process of enforcing of the constraints along with definitions of domains is the *programming* part of CP. Domains are basically sets of variables and values which they may be instantiated to. Instantiation is an assignment of a fixed single value to the variable which doesn't change further. For example the position of a taxi can be at the airport, at the hotel or at the train station (as for now we ignore that it actually moves between those positions and can be on the way). This is an example of a finite domain of a positional parameter of a taxi with $cardinality(D_{taxi\_pos}) = 3$.

4.1 CSP

A fully defined problem that we encode using the above mentioned tools is called a Constraint Satistfaction Problem (CSP). Mathematically the CSP $\mathcal{P}$ is a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a vector of variables $\mathcal{X} = \langle x_0, x_1, \ldots, x_n \rangle$, $\mathcal{D}$ is a n-tuple of finite domains $\mathcal{D} = \langle D_0, D_1, \ldots, D_n \rangle$ such that $x_a \in D_a$ and $\mathcal{C}$ is a t-tuple of constraints $\mathcal{C} = \langle C_0, C_1, \ldots, C_t \rangle$. A constraint C_j is a

tuple $C_j = \langle R_{S_j}, S_j \rangle$ where S_j si a *scope* of the constraint, meaning the set of variables the constraint affects – thus $S_j \subseteq \{\mathcal{X}\}$, and R_{S_j} is a relation on S_j, meaning $R_{S_j} \subseteq \left(\underset{\forall i}{\times} D_i : x_i \in S_j \right)$. In other words, R_{S_j} defines valid combinations of values that cab be assigned to the variables.

A solution to the CSP is an n-tuple $A = \langle a_0, a_1, \ldots, a_n \rangle$, where $a_i \in D_i$ and $\forall C_j \in \mathcal{C}$ is satisfied. Satisfaction of C_j means that R_{S_j} holds on the projection of A onto the scope S_j. In the general CSP we might want to know whether a solution exists. If it doesn't (the set of solutions is empty), the CSP is unsatisfiable.

4.2 Search

To find a solution we must *find* the solution through a *search*. The most basic search we can consider is a basic backtracking search (BT). The naive backtracking algorithm is the most elementary case and a base of all the more sophisticated algorithms. It can be imagined as a depth-first traversal of a search tree. Visiting a node means its generation and further branching is defined as picking an uninstantiated variable and assigning a value to it. At the node visit a constraint check is performed on the constraints that have no uninstantiated variables in their scopes. If the constraint check fails, the algorithm backtracks to the last assigned variable and tries to assign other values to the unassigned variable or backtracks further on.

The decision taken at the node used for branching can however bear a different meaning according to the definition of a *branching strategy*. In the basic example the node represents a set of assignments $node = \{x_0 = a_0, \ldots, x_i = a_i\}$. The branches are then extensions of this assignment set, while creating new nodes for all assignable values $\forall A \in D_j : node_{next} = \{x_0 = a_0, \ldots, x_i = a_i, x_{i+1} = A\}$. There are other possibilities though. For example a method of *binary choice points*, where two constraints are posted effectively assigning a value $(x = a)$ and prohibiting an assignment of a value $(x \neq a)$ to the variable. Another popular is *domain splitting*, where the domain of a chosen variable x_j is splitted into two or more disjunctive parts. Mathematically

$$D_{next} : D_{next} \subseteq D_j, \bigcap_{\forall next} D_{next} = \varnothing, \bigcup_{\forall next} D_{next} = D_j$$

None of these methods is universally superior and we had to choose and modify the search procedure according to our needs. A question that must inevitably arise is how to choose the right variables that we want to instatiate. This question is tackled by the *variable ordering heuristics*. It tells us which variable we might want to instantiate in order to find a solution as fast as possible or to make the search tree as small as possible. The most basic case which we followed during our definition above is a random variable selection.

We leave it on the solver which one to choose. Random variable ordering heuristic can be very useful especially in cases where we search for the best solutions by the means of restarts (after a certain threshold is reached we run the solver again, thus preventing it from getting stuck in the local minimum/maximum). However this is not always the best idea especially when we have the information about the nature of the problem.

If we reject the idea of a random variable selection, there are several principles to be followed when determining which variable should be selected. The two categories that are available are - variable ordering heuristic based on domain size and custom heuristic based on the structure of the CSP.

As for the variable ordering heuristics based on the domain size, there is a *first-fail* principle which we can use. While sounding a bit odd as we want to find a solution and not to fail, we must realise that on fail the tree is pruned and we do not explore that branch later on. The sooner we fail, the sooner we can backtrack and thus recover from the bad choice we might have made earlier on. As all of the variables must be instantiated, the variables with the smallest domain are the hardest to instantiate and thus are bound to fail earlier. This heuristic is called *Dom* as it uses the sizes of *domains* to choose the best variable.

The hardness of the instantiation may not however lie only in the domain size. As it may occur, the degrees of constraint of a variable can also be taken into account. The degree of a variable can be determined as a number of constraints that are bound to it and thus the effect it has on the CSP. Again, the more costraints there are, the more likely the variable is to fail on instantiation. There is a variable ordering heuristic based only on those degrees and is called *Deg*. Combination of the two previously mentioned heuristics is also acceptable. The combination being denoted *Dom+Deg* first chooses the variable with the smallest domain and in case of a tie uses the degree as a tiebreaker. Another combination is *Dom/Deg* which is being computed as $rank_{Dom/Deg} = \frac{Dom}{Deg}$, incorporating the degree in the primary variable choice.

As an extension, the degrees can be weighted by the weight of a constraint. The weight of a constraint is initially set to 1, and is incremented every time the constraint is responsible for a failure. This way the algorithm 'learns' during the search and it has been proven that this method is quite effective in solving certain classes of problems.

Finally, we should also mention the *value ordering heuristics*. Value ordering heuristics are basically heuristics that choose which value should be selected first when we have decided which variable we want to instantiate. They try to follow the *best-first* (or *survivors-first*) principle, selecting the value which is the most likely to survive and lead us to the solution. There are some static techniques researched [12] based on the constraint graph relaxation and then the estimation of the number solutions in which they can be present. Ginsberg et al. [12][13] has also proposed a dynamic value ordering heuristic which incorporates the sizes of domains after the application of the

value. The best value is the least restrictive, thus maximising the product of the domains in the next step.

All those techniques used in search are useful, but none of them is universal, so we have to hand-pick the one most suitable for our task or use the ones that are available through the technology we use.

4.3 Consistency

The search is a powerful tool to solve problems, however so far we have been mostly ignoring (or we have used them in a hidden manner) one of the most powerful features we have in our possession in CP - the constraint set. The constraint set is used intensively during the search to filter domains through *consistency techniques*. The consistency forces the domains to omit values which would lead to dissatisfaction of the constraints. This way the search tree is pruned so the algorithm doesn't have to construct the nodes using constraint-wise invalid values.

There are several levels of consistency, the simplest one being the *node consistency*. Node consistency is focused on the unary constraints, which it forces to be satisfied. Each unary constraint is related to just one variable by definition, so the filtering of the domain according to the constraint is a trivial one.

```
NC(node)
   foreach UnaryConstraint unary in node
      if unary is inconsistent with value X
         remove X from the domain
      endif
   endfor
```

One of the most used consistencies is an *arc consistency*. It is popular for its simplicity and expressiveness, while retaining very low computational cost. A binary constraint creates an arc in the constraint graph, thus the name. An arc is arc consistent if all the values from one domain have a corresponding pair value according to the relation of the constraint. Formally

$$\begin{aligned} C_{(x_i,x_j)} \text{ is arc consistent } \iff & \forall v \in D_i, \exists v' \in D_j \colon (v, v') \in R_S \\ & \wedge \forall v' \in D_j, \exists v \in D_i \colon (v, v') \in R_S \end{aligned}$$

Where $C_{(x_i,x_j)}$ is a binary constraint with x_i and x_j within its scope S and R_S is a relation of valid tuples defined by the constraint C. If there are some values either from D_i or D_j that are breaking the arc consistency, we can safely remove them. This can of course break the arc consistency of some

other constraints, so we need an intelligent algorithm to make all constraints consistent. This will be examined in the subsection 4.4

One of the basic consistencies worth mentioning is also the bound consistency. It is not as strict as arc consistency and enables quicker propagation and consistency checking. The idea is that every value doesn't have to have a support value from the domain of the arc-consistency-like bound variable, but only the bounds are checked. The bounds - the maximum and minimum - are however checked and all values in between are considered valid. In the definition below we suppose the construction of the domain bounds from the finite discrete domains we have used previously. The domain for the arcs with bound consistency can be also defined purely as the bounds.

$$\begin{aligned} C_{(x_i,x_j)} \text{ is bound consistent } \iff & \forall v \in D_i : b_{max} \geqslant v \wedge b_{max} \in D_i \\ & \wedge \forall v \in D_i : b_{min} \leqslant v \wedge b_{min} \in D_i \\ & \wedge \forall v' \in D_j : b'_{max} \geqslant v' \wedge b'_{max} \in D_j \\ & \wedge \forall v' \in D_j : b'_{min} \leqslant v' \wedge b'_{min} \in D_j \\ & \wedge \forall bc \in \{b_{max}, b_{min}\}, \exists val : b'_{min} \leqslant val \leqslant b'_{max} \wedge (bc, val) \in R_S \\ & \wedge \forall bc' \in \{b'_{max}, b'_{min}\}, \exists val : b_{min} \leqslant val \leqslant b_{max} \wedge (val, bc') \in R_S \end{aligned}$$

We can see that we have lost precision, but gained speed because of the omission of the intermediate values. To make the constraint bound consistent, we must update the bounds so the definition would hold. The bound update however can only make the interval shrink, because otherwise it would be the equivalent of adding values to the discrete domain while enabling previously invalidated options.

There are many more consistency types with different strengths and some are more useful than others, but their application is very domain specific.

4.4 Constraint propagation

There is a wide range of algorithms that enforce the consistency through constraint propagation with different strengths and different computational expenses. The core of the constraint propagation is a fix-point algorithm. The idea of the fix-point algorithm is that at every node, the domains are filtered until no unsupported or invalid value is present in the domains. Thus while backtracking we are always sure, that the node we are exploring from is valid and requires no further propagation.

The implementation of the fix-point algorithm differs by the strengths of the constraints that are used. In the OSCAR solver there are different levels and the constraints may implement different propagation strengths. Usually the higher strength is bought out with lower speed and vice versa. In OSCAR,

Strong constraint propagation asks for domain consistency while Medium asks for a faster bound consistency.

As it might occur, the consistency filtering of constraints may break consistency of other constraints (and it usually does). There are several algorithms that can manage the constraints during the fix-point algorithm run. The very basic algorithm is a naive one. Firstly it adds all the constraints in the processing queue. Then in proceeds with the consistency checking and domain filtering on the constraints in the queue. Every time there is an inconsistency it adds all the constraints back to the queue.

4.4.1 AC-1

The naive algorithm (also called AC-1) is very demanding and ineffective. It would suffice only for the most basic problems. When there is a lot of constraints in the model, checking all of them can take some time (especially if the domains are big as well and arc consistency is required). As the binary constraints have only two variables in their scope, the probability that it will be really affected by the current change is low.

The complexity of the AC-1 is mediocre. For every change all constraints are added back to the queue (an improvement can be introduced with the queue being interchanged by a set). In the worst case we could filter the domain values one by one from each domain in a chain-like manner. If we would add all constraints to the queue again, it would mean that we would have to check $O(d^3 * n * c)$ possibilities in the worst case, where d is a maximal domain size, n is a number of variables and c is a number of constraints. We take the d^3, because there will be $d * n$ propagations and each propagation will take d^2 operations in the worst case (see the make_consistent method).

4.4.2 AC-3

The suboptimality of the AC-1 algorithm is why the AC-3 algorithm was developed. The AC-3 adds back only the constraints which have the variable of the domain, which has been updated, in their scope. The algorithm is described on the figure 4.2.

We can see that the algorithm is much more optimal as only the possibly influenced variables are added back into the domains. Although the propagation may require more of algorithmic work, it is still quite simple and asymptotically it is much more efficient. The worst computed complexity of AC-3 is $O(nd^3)$ and space complexity is $O(e + nd)$. [3]

4.4.3 AC-4

The AC-4 algorithm offers even more refinement in the propagation using a different schema with support structures. The main idea is that not the whole

```
function AC-1(Constraints A, Variables V){
    // satisfy unary constraints

   Q <- push all c(x1, x2) from A;
   while(!Q.empty()){
      c(x1, x2) <- Q.pop()
      if(make_consistent(c(x1, x2)) {
            Q <- push all c(x1, x2) from A;
      }
}

funtion make_consistent(Constraint c(x1, x2)){
   let Domain1 <- domain(x1)
   let Domain2 <- domain(x2)
   update <- false
   forEach v in Domain1{
      if not exists v' in Domain2 such that c(x1<-v,x2<-v') is consistent {
         remove v from Domain1
         update <- true
      }
   }

   forEach v in Domain2{
      if not exists v' in Domain1 such that c(x1<-v',x2<-v) is consistent {
         remove v from Domain2
         update <- true
      }
   }
   return update
}
```

Figure 4.1: The AC-1 alorithm as presented in [2]

constraints must be placed on the queue, but just pairs of values and variables from which those values are removed. The support structures include the sets of the support variable-value pairs and counters which record how many support pairs are present. The deletion and following propagation starts only if the counter of supports has reached 0 and even then the deletion affects only one value in the variable.

The support structures are however an overhead and it depends on the problem whether they are advantageous or not. Some researchers are sceptical about the AC-4 algorithm and they empirically prove that the AC-3 outperforms AC-4 on a wide range of problems [14]. However, as the AC-4

```
function AC-3(Constraints A, Variables V){
   // satisfy unary constraints

   Q <- push all c(x1, x2) from A;
   while(!Q.empty()){
      c(x1, x2) <- Q.pop()
      if(make_consistent(c(x1, x2)) {
         forEach v from V
            if(c(v,x2) in A)
               Q.push(c(v,x2))
            if(c(x1, v) in A)
               Q.push(c(x1,v))
      }
}

funtion make_consistent(Constraint c(x1, x2)){
   ... defined as above
}
```

Figure 4.2: The AC-3 alorithm as presented in [2]

algorithm is a part of the Choco solver propagation technique and there are many of its followers which are even more specific and more effective in the means of complexity (AC-5, AC-6, AC-7. AC-8, AC-2001 and many more), we simply can't avoid it.

The derived algorithms usually introduce new structures or remove the old, making the algorithm more complex and robust to the constraint propagation. For example the AC-6 algorithm reduces the information stored to only one support for each value and when this support is removed it *searches* for another one in an ordered manner instead of memorizing all of them as was the case in the AC-4 [15]. The AC-2001 or AC-2001/3 works with the framework of the AC-3 algorithm and fine-grains the *revision* stage, represented on the figure 4.2 by the *make_consistent* method [16].

The computational complexity of the AC-4 is $O(ed^2)$ with space complexity being $O(ed^2)$. [3] The algorithm is presented on the figure 4.3.

4.4.4 Path consistency

There is a special degree, stronger than the arc consistency, that is also widely used for the definition of consistency in CP and that is the *path consistency*. Path consistency, also called K-consistency, is a consistency that incorporates K variables in their effect. A constraint graph is K-consistent if for any K-1 variables which satisfy the constraints upon these variables we can select a

Kth variable. If for each value from the last selected Kth variable there exists a combination of values for the K-1 variables that makes all the constraints upon these K variables consistent, then the constraint graph is K-consistent. We can see that every K-consistent constraint graph must also be J-consistent, where $J \leqslant K$. [2]

Path consistencies also have their specific propagation algorithms, which closely resemble their AC counterpart. For example the PC-4 algorithm follows the same schema as the AC-4 algorithm we have described [17]. Again, the propagation algorithms reason between time and space complexities, reasoning their effectiveness by empirical experiments. The PC-7 algorithm minimises the space complexity while increasing the time complexity [17]. The choice of the optimal propagation algorithm would be again problem dependent.

4.4.5 General case

The above mentioned algorithms are used to effectively maintain arc consistency defined by the constraint set. However we are not only limited to the arc constraints. The arc constraints are the very basic constraints that are used in the modelling and they are quite expressive for the basic problems. However having only them, we wouldn't be able to model everything we need. They affect only two variables in their scope, but in reality the world is much more complex. For example - how about a ternary constraint, which would say that the sum of the three can't be more than 5? We could decompose this relation into four binary constraints and one dummy constraint, however if even more variables are in some kind of relation, the decomposition is ineffective and there exist algorithms that solve the constraint more efficiently [18].

The types of constraints we can encounter and utilise are mentioned in the section 4.5, however it is good to mention them now as we have just explored algorithms for maintaining arc consistency. The n-ary constraints are obviously not covered by those algorithms. There exist algorithms which deal with the n-ary constraints and the basic ones are usually generalisations of their arc consistency counterparts.

The easiest to imagine is the General Arc Consistency algorithm GAC-3, based on the AC-3 algorithm. The AC-3 algorithm adds a constraint to the queue if there is a change in its scope caused by a propagation of a different constraint. The GAC-3 is just that, without the specification of the arity of the constraints involved. The detailed description of the algorithm can be seen for example in the lecture notes of the CP lecture of Charles University [19]. We will not go into details here as the idea is the same as in AC-3.

There are many general arc consistency algorithms, but we narrow the scope to the one that is used by our technology, which is the Seven queues propagator. [20] This will however be explained in the technological part of our solution (section 5.1).

4.5 The modelling toolkit

We have talked about the constraint propagation and search techniques and should have by now an idea how a solution is found and extracted from the world we have modelled. We have mentioned a few constraint types which are used in the modelling process, but we haven't mentioned the concrete constraints which can be used to model the world and how.

The constraints we utilize are constraints with finite integer domains. Boolean values are represented by variables with a domain $D_{boolean} = \{0, 1\}$, although they might get handled differently, depending on the implementation.

Firstly, we have the unary constraints. These are the constraints that limit the domain of a variable while not interacting with any other. The propagation is thus easy. An example would be this $(x_1 < 21)$ where x_1 is a variable. This simply limits the domain to have values smaller than 21.

Secondly, we can take a look at binary constraints. These are much more interesting, as there exist all kinds of relations that can exist between variables. If we explore the *arithmetic constraints*, we can find the following examples.

- $(x_1 \leqslant x_2)$
- $(x_1 = x_2)$
- $(x_1 \neq x_2)$

All of the constraints are quite intuitive and it is clear how the values will be checked for consistency between variables. It is quite important to note, that now we are dealing with arithmetic relations between variables, so during the propagation there may arise conflicts.

The arithmetic constraints can be effectively chained together, so in many solvers we can see constraints like $(x_1+x_2 = 5)$ or more complicated $(x_1 + x_2 < x_3)$. The first example still belongs to the binary constraints and the propagation is trivial. The second one however requires support of 3-ary constraints or global scalar constraints.

4.5.1 Global constraints

There exist constraints which don't have limits on the number of variables in their scope. They are called *global constraints* and are quite specific and interesting. Until now the constraints we have presented were very easy to grasp and their satisfaction and pruning was trivial. The global constraints are different. Every one of them express a different relation between variables and this relation can be expressed and checked by different algorithms.

First global constraint widely used as an example is the *allDifferent* or *allDiff* constraint. The variables that are recorded in the scope of the constraint all have to have different values assigned. Now, the check of the constraint for the values assigned is easy. How to express the problem so we could use the constraint for pruning and how to quickly find an information that the constraint is unsatisfiable before all variables are instantiated?

We must realise that although the constraint can be decomposed into $!n$ arcs, we lose a lot of information if we do so [21]. That is not even considering the unsupportable explosion of constraints that would have to be generated for a big n, where n is the number of variables in the scope. Thus the decomposition is a possible, yet unfeasible method for the *allDifferent* constraint.

The trick for the propagation of this constraint is its representation as a pairing problem between the values and the variables. By definition the one node set of the bipartite graph is variables and the other represents values. The edges in this bipartite graph are the possible assignments of values to variables. The detection of an inconstistency is done through the finding of a valid maximal pairing. If there is no valid maximal pairing, the constraint is unsatisfiable and the search can safely backtrack.

More complicated is the propagation. There exist different propagators with different strengths (forward checking, bound consistent, domain consistent) [21][22] , but if we were to continue with the pairing representation, we can start with the maximal matching M which we found while checking for the constraint consistency. For this pairing we orientate the edges - the edges in M we orientate from variables to values, the rest go from values to variables.

Now, the edges belong to a maximal matching iff they belong to the maximal matching M *or* they belong to an alterning cycle *or* they belong to an even length alterning path starting at an uncovered node. All edges that do not belong to any of these groups are inconsistent and can be removed, effectively applying the pruning, as the deletion of an edge is a removal of a value from a domain.

The allDifferent constraint is just one of the many constraints which can be effectively represented as a combinatorial subproblem, which have different propagators with different strengths. Constraints which are also possible are the *knapsack* constraint, which allows a definition of a knapsack subproblem on the variables of the problem, the *sum* constraint, the *global cardinality* (GCC) constraint, stating maximal and minimal numbers of appearances of values in the sets of variables, *circuit* constraint which ensures that the variables create a circuit and many more. There exist global constraint catalogs with consistent definitions of behaviour of teh constraints (for example at `http://sofdem.github.io/gccat/gccat/titlepage.html`), but it depends on the solvers which constraints they have resources to implement.

There is one important note to be said after the global constraints have been explained. It is strongly connected with the design of the model. As we have seen, the allDifferent constraint has got a domain consistency level

propagation based on the construction of maximal pairing and its extension. It has been noted that there exist different propagation levels with different complexities (for example the forward checking applies the filtering only if a variable from the scope is instantiated). It is thus of the utmost importance to have the complexities in mind during the design and to use the more effective constraints if possible or less demanding propagations if precision is not needed, as some constraints are NP-hard and don't have an efficient propagation algorithm (the *sum* constraint is a good example). [23]

4.5.2 Logical constraints and reification

If we want to model the problem as a set of boolean logical formulas, we are free to do so with the tools we have for the logical constraints. Apart from the different relations that are present in the binary constraints, the logical formulas very much resemble the models we have for the arithmetic constraints. There is however a feature that is quite powerful and that allows us to model another scope in the constraint programming.

So far we have defined the constraints as some rules that *must* be satisfied. Those are so called *hard constraints* and if there is an inconsistency, we declare the solution unsatisfiable and backtrack from the algorithm as a failure. This however doesn't cover the cases, where we want to make a disjunction of constraints. A solution at hand is to make a boolean variable, which declares whether the constraint is or isn't satisfied. This process of assigning a boolean variable to a constraint is called a *reification*.

Through reification we can specify logical relations between the constraints. Then we can work with the reified constraint variables as with standard constraint values. This allows us to define *soft constraints* - constraints, which may but don't have to be satisfied. Soft constraints are quite handy, for example for an estimation of objective functions during the COP 4.6. Also we can define new hard constraints using the reified constraints, for example for a definition of a goal constraint, where exactly one of three constraints must be satisfied 4.4.

With a possible disjunction there comes the negation and implication (as implication is a disjunction in disguise), which can form the relations between constraints further (if a helicopter is on an airfield, it must land). The same goes for the equivalency. For the negation of the problem we must realise, that the boolean variable is true if the constraint is *entailed* - there is no way that it couldn't be unsatisfied. Similarly it is false only if the constraint is *disentailed* - if it can't be satisfied.

4.6 Constraint optimisation problem

In our domain we are almost always sure that a solution exists, except maybe for some exotic instances where we don't have enough resources to satisfy the

problem (e.g. we only have a helicopter for repairs that have to be done by ships). More interesting for us is thus the *Constraint Optimization Problem* (COP). A COP is a quadruple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{O} \rangle$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are defined as in the CSP and $\mathcal{O}$ is an objective function $\mathcal{O}\colon A \to \mathbb{R}$, which assigns a numerical value to a solution. Without any changes in the CSP structure we might state that we want to maximize/minimize the objective function, thus finding the best solution.

The easiest way to solve the COP is to explore a sequence of solutions of CSP (ideally enumerating them all) and then apply the objective function on the solutions trying to find the best. This method is highly dependent on the quality of the search. However, even when the search is well formed, enumeration of all the solutions is usually computationally impossible.

There are different methods that incorporate the objective function even in the search process. For example we could estimate an optimistic estimate of the solutions not yet instantiated and use this information to extend our backtracking search with branch-and-bound features. This way we could prune the branches of the search tree which couldn't lead us to the better solution than which we have already recorded, avoiding extra work and speeding up the search.

The implementation of the objective function may be a simple function applied on the results, however this approach would not give a chance to the solver to exploit the optimization metric during the search. We could also build a hard constraint, which would track, for example through the sum constraint, the usage of resources. Another possibility is the use of the soft constraints, which could record which or how many constraints have been used and satisfied, so the cost could be associated with them. Finally, especially for the planning CP problems, the objective function doesn't have to be defined and the quality of the solution is structural and we optimize the length of a plan through the definition of the CSP problem length.

The problem with the solution of the COP problems is that we are never sure if we have found the very best solution, unless the search is complete (or the objective function is structural) and we either find all solutions or the branches that would lead to worse solutions are effectively pruned.

```
function AC-4(Constraints A, Variables V){
   // satisfy unary constraints

   Q <- queue
   A <- constraints
   data_structures(A,Q)
   propagation(Q)

}

function data_structures(Constraints A, Queue Q) {
   forEach c(x1,x2) in A{
      forEach val in domain(x1){
       total <- 0
       forEach val' in domain(x2) {
          if (val,val') in R(x1,x2) {
             total <- total + 1
             S[(x2,val')] <- (x1, val)
          }
       }
       if(total == 0) {
          Q.push((x2,val'))
          domain(x1) <- domain(x1) - val
       } else {
          Counter[(x1,x2), val] <- total
       }
      }
   }
}

function propagation(Queue Q) {
   while(!Q.empty()) {
      (x, val) <- Q.pop()
      forEach (x2,val') in S[(x,val)] {
         Counter[(x,x2),val'] <- Counter[(x,x2),val'] - 1
         if(Counter[(x,x2),val'] == 0) {
            Q.push((x,val'))
            domain(x1) <- domain(x1) - val
         }
      }
   }
}
```

Figure 4.3: The AC-4 algorithm [3]

```
b_0 <- reif((x_0 < 0))
b_1 <- reif((x_1 < 1))
b_2 <- reif((x_2 == 100))
Goal <- (sum(b_0,b_1,b_2) == 1)
```

Figure 4.4: An example of the reification - exactly one constraint must be satisfied

Chapter 5

Technology of our solution - the Choco solver

The Choco solver [24] is a free open-source java library dedicated to constraint programming. We have chosen to use it for its simplicity of use and the Java language it is implemented in. Thanks to its open-source nature we were able to analyze the algorithms and CP features implemented and could compare it with the CP theory we have explained. We have noted that different solvers support different features and constraints and implement different propagation algorithms. The propagation algorithms could be extended or reimplemented as Choco supports definitions of user defined searches, propagations and constraint definitions along with a manual with examples.

Choco allows the definition of standard integer variables with finite domains as well as real values, although those are implemented with special constraints and routed to a different subsolver. It supports the arithmetic constraints, logical constraints and global constraints. Several other useful constraints are also implemented, like constraints used for scheduling. Reification is also a must.

An architecture of the Choco solver is very much influenced by Java and uses polymorphism and interfacing extensively. The constraints are children of an abstract class AbstractConstraint and the variables are descendants of AbstractVar. Even the IntVar type is an interface as the domain implementations differ. Constraints in general have got link to the variables they have in their scope and likewise the variables have got links to constraints which affect them along with an index, at which they are stored in the constraint.

The implementation of the integer domains may differ, based on the size of the domain and its type (whether the domain is bounded or not). There are two implementations, one being a bitset and other being an array of the values in the domains, optimizing space complexity. This implementation is in constrast with the representation used in the OSCAR solver, where the domains are represented as arrays with a special timetracking class pointer

called reversible int. [23]

5.1 Propagation

A large portion of the constraint programming definition is the propagation. We have presented many possible propagation techniques, most of them being specialised for arc consistency, but have also mentioned the propagation algorithms for the general case. In the end we resort to the default technique used in the Choco solver, as it has been improved with research background and the development of a better propagation algorithm is not an objective of this work.

The propagation architecture is based on generating four kinds of events - instantiate, removal, incinf and decup. Their meaning is - instantiation of a variable, removal of a value from a variable, increase of the lower bound and decrease of the upper bound respectively. They are generated by the variables and put to different propagation structures, which serve as propagation queues in a way. The removal events have got their FIFO queue, the incinf and decup are put in their own FIFO queue and the instantiate events are put on a stack.

The solver uses a priority propagator with seven queues (or eight queues, if we trust the code more than the description). It differentiates seven levels of priority for the constraints by their propagation complexity. The levels are the following:

- *level 0:* instantiate events
- *level 1:* removal events
- *level 2:* incinf and decup events
- *level 3:* feasible tuples are propagated (AC-4)
- *level 4:* linear constraints are propagated
- *level 5:* global constraints with subquadratic complexity are propagated
- *level 6:* global constraints with quadratic complexity are propagated
- *level 7:* all other constraints are propagated

During the propagation the algorithm processes events from the queue with the lowest level and moves on only if the queue has been emptied. The logic behind this is that the events from the queue with lower levels are much quicker to be processed. Also, considering the speed, if we would call the complex constraints often, the time would rise notably while the significance of the propagation would be small.

To reduce the processing time of the more complex constraints, the Choco solver also introduces batching of the events into one abstract event for the constraints above level 3. This allows the global constraints to process the

events indifferently on the amount of changes that were introduced in the previous layers. As they are delayed anyway and the number of domain changes can be high, this improves the performance of the propagation. Each constraint however defines its own abstract events, as their propagation may depend on the types of event they consist of.

Choco solver also implements a mechanism that prevents duplicate or inferior events being queued into the queues and alternatively it can also strengthen the events in its propagator queue (for example if an event doesn't only update the lower bound but accidentally instantiates the variable).

It also implements a special pointer system which should prevent unnecessary wake ups of the constraints. Firstly they don't wake up the constraint that is the initiator of the event. Secondly they store a special array of pointers by which the variables know which constraints should be awoken, as some might be satisfied even after the event processing takes place (for example a constraint that observes the upper bound doesn't have to be woken up on the lower bound or removal update).

5.2 Search & other features

The search in Choco is fully modifiable and Choco also comes with a set of predefined search strategies ranging from DomOverWdeg to activity based search.

Choco also provides monitors, which basically monitor and record the events that happen during the search. These can be initialization, opening of a new search node, when a solution is found, while backtracking etc.

If the monitoring is not enough, we can also use the large neighborhood search to make the algorithm search through the neighbors of the solution found. The main principle of the LNS is a *relaxation* of the problem. The definition of relaxation in this context is an un-constrainment of the domain of a selected variable or a set of selected variables. The algorithm takes the (best) solution, relaxes a few variables creating a partial solution and then searches for the complete instantiations of the partial solution. The idea is that some better solutions may be present around our best one, however through the tree structure of the main search tree it would be difficult to reach them.

Choco provides a LNS factory, through which we can define the variables to be relaxed, a number of them which will be fixed, the number of failures on which the algorithm will cease searching or another metric like time or backtrack count.

Chapter 6

Research for our solution

Now that we have the tools to work with researched and basic methods defined, we need to start defining and implementing the problem we have chosen to implement. It has been decided that we will start implementing the parts of the system and research the technologies on the go. A technological know-how was needed when the implementation started, so it would have been possible to make a development plan, however it is better to present the technologies and their possible alternatives or extensions as we reach the appropriate implementation points.

6.0.1 Problem type identification

We have outlined the problem we are solving in the section 1. Along with the PDDL specification there is a clear idea of what is needed to model and what are the optimization criteria. We know how to construct models for the classical CP problems. The domain into which our problem belongs is however halfway between a scheduling problem and a planning problem.

From scheduling we take the timing of actions as we need to create a plan least costly for the company. Also, we utilize resources, for example engineers, which are needed to execute the repairs on the wind turbines. The resources also influence the feasibility of helicopter action planning, as the helicopter can run out of fuel.

In scheduling however we usually know the set of actions in advance and our objective is to *schedule* them in an efficient way. In our case we are not sure which actions will be needed in order to satisfy our goal. This is where planning comes into play, as it is exactly the choice of actions which we will need to reach the goal state from an initial setup.

The CP planning however is a bit more difficult to model, so it is a good idea to start with it. For scheduling there already is a large set of constraints and task structures which altogether help with the modelling, so the main focus is set on the CP planning with time extensions for durative actions.

6.1 CP planning starting point - the Graphplan

The planning problems are problems, where we have the world and its properties defined as well as the starting state of the world, the goal conditions which must be satisfied in the end and a set of actions, which transform the world from state A to state B. The objective in the planning problems is to find which actions to apply in which order for us to reach the goal state from the initial state.

The starting point for us is the Graphplan algorithm [9]. Graphlan is a solver for planning tasks with problems defined by the STRIPS domain language. We have already mentioned STRIPS, as it is the predecessor of the PDDL language. Graphplan constructs a planning graph, which the authors define as a directed graph with two kinds of nodes in layers and three kinds of edges. The *proposition nodes* are labelled with logical propositions and they are kept in proposition layers. The *action nodes* represent the actions and are organized in action layers. There are precondition edges from preconditions to actions, which determine the requirements for the applicability of the action. An action can be activated if all its preconditions are true. There are also add-effect and remove-effect edges, going from action nodes to precondition nodes, which trace the effects of the action on the following precondition layer. Special no-op actions are present to transfer the propositions which have not been invalidated into the following layer.

There is a definition for the limitation of parallelism of the actions, so as no two actions can happen at one time step if there is a mutex relation between them - if they don't interact with each other. Two actions are in a mutex relation iff one deletes a precondition or an add-effect of the other. Similar rules apply to the conflicting prepositions.

A valid plan is a set of actions decorated with times at which they are carried out.

The algorithm then constructs a plan of a minimal length and tries to find a solution through the search in the graph. If a solution is not found, the graph is then extended by two extra layers and corresponding constraints, allowing a search for graphs of a greater length. The process is repeated until a solution is found.

6.1.1 Graphplan CSP - CPlan

The planning graph can be compiled to the CSP and solved by the CSP solvers. One of the approaches is presented by the CPlan system [25] , where they decompose the planning graph into a set of constraints and variables, performing better than prior solvers (like Blackbox) [25] , which use the decomposition of a planning graph into a CNF formula and then they solve the problem as a SAT problem. However it also expects to have all the constraints hand-picked and hardcoded.

6.1.2 Graphplan CSP - GP-CSP

The GP-CSP planner [26] presents an automatic scheme to encode the planning graph into the CSP problem. It defines the mapping between the Graphplan constraints and CSP constraints as well as the mapping from the Graphplan propositions and actions to CSP variables and their domains. As an advantage from the conversion we get an applicability of different search techniques, so a non-directional search can be used (or any defined so to say), in contrast to the simple backward search in the graphplan. Also the GP-CSP scheme introduces additional techniques like reachability analysis and constraint analysis, as most of the mutex constraints in the Graphplan are derived and therefore redundant.

In the encoding of the GP-CSP solver, the propositions are mapped into the CSP variables and the actions are assigned CSP value numbers. The propositions then have the actions which make them true in their domain along with a null value. The constraints are encoded as functions, which can check their satisfiability through the values of the variables they are assigned to, and they are organized in a global hashtable and linked with their corresponding action. Also based on their value, they state whether some predicates in the previous layers are *active* and therefore must be set by an action sometime before.

However, we don't have variables for actions and there is a huge amount of constraints for every original action mutex constraint. To be precise, for every precondition of actions in mutex relation, every combination of variables instantiated into values corresponding to the mutex actions must be constrained by mutex.

$$\forall a_1, a_2 \in Actions, \forall p_{11} \in Support_{a_1}, \forall p_{12} \in Support_{a_2}, mutex(a_1, a_2):$$
$$\neg(p_{11} = a_1 \wedge p_{12} = a_2)$$

The effect of modelling the graphplan without action variables is not very comfortable and further research has been done to improve the modelling methods.

6.1.3 Graphplan CSP - CSP-Plan

Based on the GP-CSP and CPlan more sophisticated CSP models representing the Graphplan have been created. One of those is the CSP-Plan implementation [27]. It uses the action variables along with the preposition variables, all organized in layers as in the Graphplan algorithm. Also the iterative search procedure, where we create a plan of a minimal length and then increase the size on complete search fail, is retained.

In order to retain the consistency in propositional layers, the Reiter's successor state axiom is used. [27] In simple words it states that if a propositional

variable P_j is true in the layer L, it has either been made true by an action in the current action layer or it was true in the previous propositional layer and hasn't been deleted by any action in the action layer.

$$P_j^L \iff (P_j^{L-1} \wedge \bigwedge_{a \in ActionDelete_{P_j}} \neg a) \vee (\neg P_j^{L-1} \wedge \bigvee_{a \in ActionCreate_{P_j}} a)$$

The successor state axiom guarantees the consistency between time frames.

The constraints used are mostly the same as before - the preposition mutexes, the action mutexes and more. However the authors have explored a stronger consistency, the *2-j* consistency, to derive even more constraints. A technique to make the problem easier to solve is to create redundant constraints, i.e. constraints, which are not necessary for finding a valid plan, but which additionally reduce the domain, therefore speeding up the search. Additional constraints are thus useful.[27]

The *2-j* consistency is a strong one - it states that for every assignment to 2 variables there is a set of values for j additional variables such that the *2+j* assignments satisfy all the constraints over these variables. We can see some similarities with the arc consistency, but with much more general strength and with corresponding computational cost. However the authors propose the $2-2$ variant for the definition of additional action mutex constraints in a following way - if there is a mutex constraint between the propositions P_i^L and P_j^L and there are actions A_i^L and A_j^L with P_i^L and P_j^L as their preconditions, the *2-2* encovers a new constraint $\neg(A_i^L \wedge A_j^L)$.

Other new constraints are derived propositional mutexes. To construct the new propositional mutexes, the actions must be inspected and then, based on the three scenarios, it must be proven that there doesn't exist an action that would make the propositions both true. The three scenarios are defined as situations, where the propositions P_i and P_j are in a state $\neg P_i \wedge P_j$, $\neg P_j \wedge P_i$, $\neg P_i \wedge \neg P_j$ and a special case $P_i \wedge P_j$. The preconditions of the mutexes are then checked through the enumerations of the actions and the proof of their validity is therefore constructive. The special case $P_i \wedge P_j$ is applicable only when those propositions are present exclusively in the first propositional layer.

It is necessary to note after the definition of these new constraints that the null action layers (the layers where no action is active) are prohibited in the model proposed as all of the constraints would have to cover situations after null actions.

Apart from the new constraints based on the Graphplan CSP the authors propose new techniques beyond the Graphplan CSP representation. One of the easiest propositions is the Symbolic Reduction of Single Valued Variables. The technique simply eliminates the variables present in the CSP model and

replaces them with constants, thus simplifying the problem. Because all constraints and variables can be translated into logical formulas, some simplifications are straightforward ($P_j \vee false \Rightarrow P_j$). The same applies for special cases, where there are no adders or deletors of the propositions among the set of actions - they can be reduced to a constant.

Another new binary constraints can be derived by the same analysis that was used for the derived propositional mutexes. Before we were only restricting the use of the ($TRUE.TRUE$) combination, however using a similar reasoning we can derive the same constraints for the negations of the variables, forcing their values to be both true or both false.

Finally, custom constraints are proposed, namely widely used sequence constraints which restrict the use of the inverse actions in sequence. The search therefore doesn't have to search through the possibilties, where we do an action and then do its inverse while not making any change in the propositional world. Those constraints may be useful, however we should be careful with their use with the temporal actions (as they are not taken in the consideration now), because the inverse actions may cause some change in the CSP modelled world with times and resources (e.g. we want to fly the helicopter until the fuel tank is empty).

6.1.4 Graphplan CSP - Reformulation of the CSP

There exist many ways to represent the same CSP, some efficient, some less so. There are attempts for reformulation of the constraint model through the simplification of the constraints and/or their reduction and representation with more expressive simpler constraints with better pruning or control. One of those optimizations is a substitution of multiple constraints by a simpler table constraint [28], [29]. The table constraint is a constraint over a n-tuple of variables, to which we can define a set of tuples of values, which are valid (or alternatively invalid, if the modelling tool allows). The idea is to group similar actions into one table.

The graphplan is modelled in a similar fashion to the CSP-Plan proposition or the GP-CSP proposition (actually multiple models are considered), although there is always only one action variable in the action layer with action IDs in its domain. This has its advantages, as we don't have to consider the mutexes between actions and the parallelism can be introduced afterwards on the solution found. Also the sequence of actions is very easy to implement and so is the frame axiom or successor axiom, as we are dealing with the effects of one action at a time.

However this approach also has its limitations, as the missing parallelism can be considered an issue, because a parallel plan, which would be able to be satisfied in one steps with n parallel actions, would have a length n in the serial approach. This however is a choice of the implementor as both approaches have their pros and cons.

The reformulation of the CSP using table constraints is demonstrated on the serial plan, however it could be extended (with certain limitations) into the parallel version.

6.1.5 Graphplan CSP - Reformulation of the CSP II

Another reformulation of the constraint model comes from the same author [30]. This time however the reformulation is not concentrating on the model, but rather on the search techniques. We have not considered the search techniques much so far, as the previously mentioned models used more or less the searches that we have already mentioned in section 4.2, or the authors state that the search is problem dependent and that we should choose the search according to our needs (and empirical evidence) as all problems are different.

For example in the previous reformulated CSP model the search was a simple static variable ordering heuristic instantiating actions from end to start. Note that it is advantageous to make only the action values the decision variables as the rest is either not important or, more often, is fully determined by the choice of the actions.

However, the search tehcnique the authors propose is a domain splitting technique called *lifting*, where the actions are split into groups according to their effect and preconditional domains. This allows the backward search to postpone the decision of the concrete action until later, however the knowledge of influenced variables remains.

The authors also introduce the symmetry breaking constraints (or Dominance constraints), which define an ordering between the non-interfering actions. This is crucial to the model presented as no parallelism was allowed, so if there was n potentially parallel non-interfering actions, there was $!n$ possible orderings, which were all valid and functionally equivalent.

The last improvement introduced is a stronger consistency defined.

6.2 Durative actions

One of the most problematic issues we deal with in the implementation is the representation of durative actions. Until now the representation of actions sufficed to model only the basic planning tasks that could be very easily solved by different methods, be it the CSP approach or the chain forward iterative approach. As we use the CSP paradigm, we have to find a way to extend the planning model to include also the durations of the actions.

Firstly we have to revisit the definition of durative actions to see what features are needed and which similarities do they bear with regular actions in our CSP planning graph. Then we can research the works of the authors of the planning graph models we have mentioned above to see how they solved the issue of the duration of actions in their CSP formulations. We can also

make first guesses and notes as how those actions could be represented just exploring the structure of the planning graph we have used until now.

6.2.1 Revisiting the definition of durative actions

We have already talked about the durative actions in section 3.2.1, where they were a part of the PDDL domain definition. The definition in general stays the same, although now we have to take into consideration the implementation of those constructs. The durative actions have *preconditions*, which are generally identical with preconditions of simple actions, *effects*, which resemble the effects of simple actions, although their enforcement is not instantaneous, and *invariants* which have to hold during the whole 'run' of the action.

6.2.2 Basic notes

If we look at the previously successfully modelled planning graph, we can observe the difficulty of implementation of durative actions right away. The *preconditions* stay the same. The *effects* are also the same, although they are applied only after a certain time. This could be easily solved if every layer would represent one atomic time step - the effects would simply be joined by a constraint, which would force them to be true after certain number of steps. The *invariants* are a problem, as we have to make sure that they are satisfied throughout the whole run (endpoint excluded). If we again returned to the model of atomic-step-per-layer, this could be also easily solved, enforcing the invariant constraint on every layer separately. A boolean variable would act as an activator of the constraint and an indicator of the action in effect.

However this representation creates a lot of issues by itself. Although very easy to imagine and moderately easy to implement, there are errors which arise even before we formulate a formal definition of the extended graph. One of the issues is that we have ultimately lost connection to the durationless actions. In the standard planning graph we can also speak of timestamps of the layers, however they were used only for the correct estimation of an order of the actions and didn't say anything about the durations. Everything was, in the end, instantaneous. We *could* define that the simple actions have a duration as small as is the duration of the shortest durative action. This would however prolong the generated plans, forcing simple non-durative actions to wait for each other and actions that could be instantaneous (like purchase component) would take at least 15 minutes, as is the time of the shortest durative action. We could also pick an arbitrarily small time length like one second. This would only lead to another issue - the granularity of the graph.

The granularity is even a bigger issue in our problem. Lets consider that we would pick one minute as the most useful time to model the durations of simple actions. Then the shortest durative action would need a plan of 15 layers to be satisfied and if we would like to create a plan which would cover a

whole day, we would end up with a plan with 1440 layers, which would be very hard to solve. Taking into regard that in the basic problem instance we have a repair action which spans 184 hours, this approach as-is is clearly unusable for our further research.

Another issue, not being connected to the durative actions at first glance, are the timed initial literals (discussed in section 3.2.1). Having an effect on the actions, it would be easy to represent them in the atomic-time-step-per-layer model – they would be defined as constants present in the corresponding layers. However new approach will have to be found based on the planning model we will create. As they make the proposition hold or not based on the times in the plan, they closely resemble the durative actions with starting points and endpoints equal to their predetermined time points.

TILs could be also represented as constraints on the start/end actions. We could make them fall into a set of intervals, during which the allowable conditions hold. Of course then we would have to compute whether the action could or couldn't finish in the same interval to prevent infeasible solutions, possibly precomputing the possible starting and ending times and allowing the actions to start only when they would be able to finish. The static nature of the TILs could be easily exploited.

6.2.3 A flexible constraint model for validating plans with Durative actions

The article with the same name as the name of this section is one of the first papers related to the problem of the description of durative actions [31]. The procedure is slightly different, as it works with partially ordered plans with durative actions and thus it doesn't create the plan from scratch as is defined in our task. However the author exploits further the structure of the planning graph, enriching it in order to represent the durative actions.

The representation is inspirational for our solution as it tackles some problems foreshadowed in the notes (6.2.2). It uses a boolean planning graph enriched with numeric parts of a graph. It transforms the action variables in action layers so an action can have three boolean variables in effect - *start*, *end* and *middle*. The *start* variable indicates whether the action starts in the selected layer. The *end* variable serves as an indication for the finish of action in the layer correspondingly. The *middle* variable indicates the run of the action during which the invariant conditions must hold. Those variables are logically bound, so the start action is always entwined with the middle action, which has to be true through the action layers of the planning graph until an end action is reached.

For the interlayer consistency the successor state axiom is again used with corresponding constraints which make prepositions true when they are added by some action or when they were true in the previous layer and have not been deleted by an action.

The representation includes another feature - the times of the propo layers and times of the actions. For every action we have additional $StartTime_a$ and $EndTime_a$ variables, which store the starting and ending times of the action. If there are any action precedences, it is easily modelled through these variables. Additionally the duration can also be encoded as a constraint on these variables, as $StartTime + Duration = EndTime$.

There are also variables that interconnect the numerical and logical parts of the model. Every action has a $StartLayer_a$ and an $EndLayer_a$ variables, which contain indexes of layers at which the $Start_a$ and End_a actions are true, that is if we maintain those variables in arrays (which we usually do). This is achieved through the *element* constraint, present in almost all good solvers (and Choco is not an exception), which models exactly this relation. The *element* constraint is also used to create a link between the $Time_L$ in the propositional layer and the values of the $StartTime_a$ variables through the index determined by the value of $StartLayer_a$.

The authors then experimented with the representation and found out that the more appropriate representation is to make the solver process the boolean variables as simple finite domain problems. The method used for the boolean network (Binary Decision Diagrams) have proven to be memory expensive and made the problem crash.

As the decision variables of the problem they used the numerical variables *StartLayer*, *EndLayer*, *StartTime* and *EndTime*, as all the other variables are then positively determined by the constraint propagation.

The difference between our problem and the plan validation problem in the paper manifests itself in the simplifying constraint over all *start* variables limiting their sum to one, thus the action can take an effect only once. And not only this, the ({Start,End}, {Layer, Time}) variables are only one per action, which would make the same implication. We could make multiplication or cloning of those actions to allow more instances, but we don't know that in advance. The limitation is thus not suitable.

Another limitation implicated by the previously picked actions is the fact that we can't ignore some actions if they are not needed - every action has to have its valid time assigned. This could be solved by introducing possible null values interconnected by constraints, however as the model is unsuitable due to the reasons stated above, this modification wouldn't make it more suited to our needs.

6.2.4 Durative action decomposition

The previous model, apart from building on the aforementioned works on the Graphplan CSP algorithms, also utilises some transformations from the paper by M. Fox and D. Long [32] which desribe the modification of the planning graph to encode durative actions.

The authors start from the PDDL 2.1 description of the durative actions and then propose decomposition. It splits the action into the start action with a duration and two instantaneous actions, one representing the end of the original action and one its invariant. In order to manage the start and end point as a pair, the intermediate actions have a requirement of a proposition, that is achieved by the start action, and an effect, that is required by the end action. The invariant action must be enforced to appear on every layer between the start and end points in order to enforce the invariant check. By default the Graphplan scheme would introduce *no-ops* to persist the effect of the first invariant action. The solution is to introduce an additional proposition as the effect of the invariant required by the end action. Additionally the graphplan must be restrained from the introduction of *no-ops* for the propositions generated by the start and invariant actions.

A graphplan modification of time handling is also proposed, as the technique used was originally based on the TGP system, which prohibited parallelism of durative actions. The model of the TGP system resembles the model proposed and criticised in 6.2.2, as it uses uniform increments between the layers. Contrary to that the authors propose a model, where the propositional layer are labelled with times and are used only to capture the points where something happens. It is however also noted, that this modification causes the algorithm to lose its time optimality, as a more time-optimal solution using two actions is explored later than a solution using one long action.

The search is the component most modified. It gradually introduces constraints, that the start and end layers must be exactly *duration* away. Also there are constraints introduced that state that the layers between the start and end actions must have a duration less than the duration of the associated action. Those constraints allow the authors to formulate the problem as a linear programming instance, that can be solved by a simplex algorithm (or other LP methods), effectively minimising the total duration of the plan. The form of the constraints is kept in a matrix, with as many columns as there are layers in the plan and as many rows as there are durative actions in the current plan (so on the start there are no rows). They modify this matrix every time an end action is added (a row is added and a corresponding position is set to 1), an invariant constraint is added (corresponding position is set to 1) or the start action is added (the equality constraint is formed). Backtracking makes the matrix revert in the exact same way (positions set to zero, reformulation of the constraint or a deletion of a row).

This approach is indeed complex and there is a difficulty that has to be addressed regarding the timed initial literals. We may have troubles representing them, as they represent time intervals during which some actions can be executed. As this model uses the durations of the layers, searching for the solution through backtracking, this would mean that the TILs would be checked at the very end where all durations would be assigned and thus we would know at what times the actions are supposed to be executed.

These problems would arise if we would represent the TILs as durative actions, as we proposed in the section 6.2.2. The same would apply if we would reformulate the TILs as constraints on the start and end times of the affected actions, as, again, we are not sure about the time when the plan execution would start and what would be the exact times of their application.

6.3 The Activity-Based search

During the intermediate experimentation we have found that the idea of our search is impractical and slow. We experimented with other search variants already implemented in the Choco Solver and found one, that had in general much faster search than the forward-chaining-like alternative. The activity based search is based on recording of the activies of the variables during the search. For every variable it keeps an activity counter. The values of the counters are then used during the variable selection phase of the search. For the value selection there are also activity counters for the values, which are computed in a different way.

We will describe the rough idea of the algorithm. The mathematical formulas describing the process are very well written in the source paper [33]. The search, after every decision and subsequent propagation made, differentiates between two sets of variables. The *Passive* set contains variables whose domains did not change as a result of the propagation after the assignment. The other *Active* set contains variables which have shrunk as an effect of the current decision, usually an assignment of a value to a variable. We can see that the sets are disjoint and their union equals the total variable set X from the CSP problem definition (section 4.1).

6.3.0.1 Variable selection

After every step the activity counter is either increased by 1 iff $x \in Active$ or, if $x \in Passive \wedge |D(x)| > 1$ it is multiplied by a decay constant μ, while $0 \leqslant \mu \leqslant 1$. The condition of $|D(x)| > 1$ is in place because otherwise the variables which are instantiated early would quickly have their activity counter reduced to 0.

For the variable selection the algorithm picks the variable, which has the largest ratio $\frac{A(x)}{|D(x)|}$. The $A(x)$ is the activity counter. The division by the domain size lessens the advantage of variables with greater domains, because naturally the variables with more values in their domain should react to more external constraints. If there are multiple choices of variables with the same ratio, the winner is randomly selected between the tied ones. This method clearly follows the fail-first principle, as the most influenced variable, therefore possibly the one most difficult to instantiate, is picked.

6.3.0.2 Value selection

The algorithm also records activity counters for the values in the domains in order to define relevant value selection. The activity of a value is however defined differently. An action of a value is defined as a size of the *Active* variable set after the application of the decision which assigns the value to the variable ($A_k(x = value) = |Active|$) [33]. This activity value however differs at different nodes of the search tree, therefore an average sum over the activity values for each value in each variable domain is proposed. Because we don't have the information about the whole search tree, a weighted sum gradual scheme is proposed, which takes into the account the previous value of the counter multiplied by a coefficient and the activity value of the current node.

As for the value selection, the value with the lowest activity is selected, clearly following the best-first principle.

6.3.0.3 Initialization

The algorithm requires some initialization, as there is no information at the very start and all the activity counters are initialized to 0. The initialization therefore features *paths*, where every path of a length k is an$k$-tuple of assignments in the form of $(x = val)$ where $x \in X$ and $val \in D(x)$. The paths used for the initialization are short and randomly generated and also called *probes*. All the probes constribute to the estimation of the mean activity of the variables and the values in the same way as was described in the variable and value selection, although without the decay. The number of probes is chosen so much to provide a statistically significant estimate of the mean value – in practice it means that the probing halts when the 95% confidence interval of the t-distribution for the mean value is sufficiently small.

Chapter 7

Design and implementation

The biggest issue that was tackled was the formulation of the durative actions. Prior to that, the graphplan framework with modifications and enhancements proved useful for initial planning problem tests. This inspired us to explore the possibilities foreshadowed above and see how the implementation would work with the technology we chose to use. The main problem is that none of the approaches deal with all of the features that are present in the problem, although some could be resolved through some additional propositions, which would have to be analysed and tested for soundness and eventual efficiency.

Also there is a possibility of decomposition of the problems into a sub-problem to make the plans only for discrete parts that would be determined by, for example, resources. The resetting of the wind turbine may be very well handled by the helicopter alone and a helicopter is usually less costly to rent than a ship, taking into consideration its speed and the lease cost per hour, as the Workboat is a bit less expensive, however it faces additional limitations, which the ship doesn't have. Of course the ship also has its own limitations, but it doesn't have the constraints on the fuel reservoir and on the operation by night.

This idea of decomposition is similar to the handmade solution, as we can make a plan for each repair and then join them together, if possible, as it would decrease the total time and thus the total cost.

However, in the end we resort to the all-in-one approach, where we create a planning graph that contains all the features needed. There is a risk that the model will be too big and the search will take a very long time, as when an all-in-one approach is used, the completeness is traded off by a worse scalability of the model.

7.1 The source PDDL domain & problem - detailed description

The authors encode the domain in the WINDY-SIMPLE and WINDY-COMPLEX encodings. They are very much alike except for some details in the lease policy of the vehicles.

```
(:types component turbine ?windfarm vehicle landable ?factory - locatable
turbine port - landable
airport seaport - port
ship helicopter - vehicle
workboat heavyship - ship
cmsv barge - heavyship
barge - airport
seaport airport warehouse factory turbine windfarm port landable - location
;warehouse factory windfarm landable
gearbox blade lru - component
)
```

Figure 7.1: Type hierarchy as presented in WINDY-COMPLEX

The type hierarchy of the domain is present on the figure 7.1. The entry point to our model are the *locations*. We can see there are factories where components can be produced, warehouses which can have the components stored, airports for helicopters, seaports for boats and windfarms with turbines.

The boats are further divided into workboats and heavyboats, while a heavyboat can be either a Cmsv or a Barge. Unfortunately the authors don't mention what a Cmsv is, but we can imagine it as a ship big enough to transport bigger components, however lacking the equipment to make a repair of the wind turbine.

The turbines consist of components which can break. The blade and the Gearbox are large components which require a barge to be repaired. A LRU (Line-replaceable unit) is a small part of the wind turbine made to be easily replaceable in case of a failure and small enough to be transportable by any vehicle.

There are engineers freely available at the ports (seaports or heliports) which are needed for the on-site repairs and inspections.

7.1.1 Actions and Durative actions

There are 10 actions and 20 durative actions in the model. The actions and their meaning are listed on the table 7.1. Detailed descriptions with preconditions, effects and possible invariants are not that important to list as they are inspected later in the modelling phase.

Actions	**Note**
lease	unintrusive - leases a vehicle
purchase-component	instantaneous - makes component available to the model
disable-turbine	instantaneous – disables turbine
enable-turbine	instantaneous – enables turbine
embark-vehicle-from-port	loads one engineer from unlimited pool
embark-vehicle-from-port-bulk	loads engineers to full capacity from unlimited pool
embark-vehicle-from-turbine	loads one engineer from turbine
embark-from-turbine-bulk	loads all engineers from turbine
disembark-vehicle-to-turbine	unloads an engineer to turbine
disembark-vehicle-to-port	unloads an engineer
Durative actions	
refuel-helicopter	refuels a helicopter
reset-turbine	resets a turbine
inspect-turbine	inspects and resets a turbine
pickup-engineer	loads one engineer from turbine
enter-site-heli	while at a windfarm, fly to a turbine
enter-site-ship	while at a windfarm sail to a turbine
leave-site-heli	while at a turbine, begin flight and fly to windfarm
leave-site-ship	while at a turbine, set sail and sail to windfarm
sail-between	sail between the turbines of the same windfarm
fly-between	Fly between the turbines of the same windfarm
sail-to	link sea
fly-to	link air
load-component	loads a component on a heavy ship
unload-component	unloads a component on a location
manufacture-part	creates a working component
deliver-component	move component from A to B
replace-gearbox	replaces a gearbox
replace-blade	replaces a blade
replace-lru	replaces a LRU
transfer-to-barge	move a component from a cmsv to barge

Table 7.1: Actions and Durative actions as proposed

7.1.2 Logical inconsistencies of the source PDDL domain

Since we hardcode the domain and make the solver one-purpose, a deeper inspection of the domain and the problem specifications must have been made. This inspection has uncovered some questionable properties of the model, which result in illogical behaviour. We address these fallacies in this section and note how they could be solved or specified differently to make more sense. As the authors of the domain state that it has been made in cooperation with industry, we have to retain as much properties as possible, because we don't have any additional info at our disposal to rule which way the domain should look like in reality.

7.1.3 Domain specification - Type hierarchy

In the definition of the object types there are some relations which demand attention. As we can see on the figure 7.1, some types are added more than once in the scheme. For example a turbine is a *landable*, yet both of them appear in the *locatable* definition. This may be a PDDL requirement, however in the definitions of actions it doesn't seem that is the case. This problem is not a grave one and has no impact on the functionality.

The fact that a windfarm and fatory are *locatable* and thus may be located at some place is more questionable. Since working with those types requires no further information on their location, because the distance between their instances is encoded in the problem specification, this relation is wrong. Even the fact that the airports and seaports are not used further on except as separate entities makes us assume that the relation for them is also wrong. Finally the relation which specifies that a barge is also an airport - it is reasonable to consider the fact that a helicopter could land on a barge (and be transported elsewhere). Nevertheless this property is not used anywhere in the model and therefore the relation is redundant.

The model however doesn't require a change as these mistakes only add unimportant properties to the model which we will ignore.

7.1.4 Domain specification - Actions and durative actions

The actions are well defined. The durative actions however have several issues which are inconsistent with the specification and/or the logic of the model.

The *pickup-engineer* durative action, which represents a pickup of an engineer from a turbine by a vehicle is a durative duplicate of the instantaneous *embark-from-turbine* action. It is not logical to have it present when both of them would be available - the instantaneous action should always be used. The reason it is present in the model probably is because the authors found it reasonable to introduce some time requirements for the pickup. They may have struggled with the specification of the requirements for the action and

without it the durative action is meaningless. We therefore remove the durative action from the model definition.

The *enter-site-heli* durative action, which represents flying to a specific turbine after entering the windfarm has a different duration and the helicopter range resource consumption specified. It may be possible that the two differ, as the landing procedure may consume more fuel, but it is not consistent with all other actions. As a starting point we consider the duration and resource consumption to be equal and may propose the modification in the possible future work section.

The *fly-between* durative action, which represents a helicopter flight between two turbines of the same windfarm has a weird precondition which doesn't make sense. We consider it a typo and ignore the condition.

The *manufacture-part* durative action, which represents a manufacturing of a component is reasonable, however incompatible with the problem specifications the authors propose. The authors propose concrete components with concrete names and as the goal they have them placed on previously specified positions. This is a conflict of two levels of planning, one where the components are described by an Id and one where they are only fluents. We resort to not using this action altogether.

The *replace-lru* durative action, which represents an installment of a LRU, is applicable for all vehicles, however the loading action is present only for the barge. As the LRU is a small component, which should fit into any vehicle, this is a hole in the definition which can be solved by a definition of a dedicated action, *load-lru*. This decision can be questioned as we don't have accurate data about how big the LRUs are in reality.

7.1.5 Problem specification

The problem specification has issues mainly in the consistency of the model. As it was developed, there are some remnants which were not removed or finalized and they state obsolete or unimportant information.

The first (minor) issue is the question between the relation of the *link-land* property definition and the *time-to-deliver* propertu definition. It is not clear whether the program should report an error when there is the *link-land* declared between two locations in the instance specification, but no time is assigned. The problems are well defined in this matter, however we feel that there is redundant information in there easily breakable.

Next issue is connected to the type hierarchy mentioned above (7.1.3) regarding the airports. The airports are declared as *locatables*, so they are supposed to have a location to belong to. This is actually the case in the declaration, as (e.g. in the problem *pfile01-tn.pddl*) northheliport belongs to the northport, windfarmAheliport belongs to the windfarmA and windfarmB-heliport is present at the windfarmB. However, this structure is not used in any way in the model. The windfarmAheliport and windfarmBheliport can

never be used. This is not only an issue of unimportant information, but theoretically if those airports would be present and available at the windfarms, the usability of the helicopters would be much increased, as they could refuel on-site. We don't know whether there are resources on the windfarms or not, so we will resort to the 'harder' version where helicopters must return to the northport for fuel.

Inconsistent information is also present in the daylight specification. In the problem there are comments assuming that there are 14 hours of daylight in a day and 10 hours of darkness (actually 8 hours of darkness, but it doesn't add up). This is however not the case even in the first few daylight cycles defined where they are defined randomly (10h, 14h, 12h, 12h, ...) . To make things even worse, for some reason the definition of the daytime even switches after 326 hours and out of a sudden there are 14 hours of darkness and 8 hours of daylight. The most important inconsistency is present from the problem 4 onwards - the daylight TIL is not defined until 326th hour, making the daylight span for all those 325 hours and solving the problems without daylight restrictions.

The resolution for the daylight TIL is to make a static assumption that there are indeed 14 hours of light and 10 hours of darkness starting with the light hours and extend this constraint to the whole plan. While it is clearly a step back in the specificity of the problem, the faulty definition of the problem may produce invalid plans and this approach is less error prone. Possibly we could make the daylight cycle configurable.

7.2 Implementation overall

The program is created from 3 parts - the Parser, the Logical & Filtering layer and the Model (project). They are developed as maven java projects, linked together by the maven dependencies. In this section the basic structure and responsibilities are outlined. The more thorough descriptions of filtering and modelling is

7.2.1 The Parser

The Parser handles the load and basic conversion of the PDDL specification of the problem instance to some organized program structure. As we have the PDDL domain given and static (we don't expect it to change), it can be safely hard encoded in the model, instead of an implementation of a general parser for the domain, as it would multiply the time required to finish the problem.

The parser consists of a simple java tokenizer and an implementation of the push-down automaton. It is not very flexible and some rules have to be followed to consider the problem well-formed, however it is quick and it suffices for our needs. If the model is not well-formed, the parser reports an error in the form of a runtime exception.

It identifies the objects and stores them in a hashmap. There are 12 final object types altogether with 9 possible interfaces to implement. The hierarchy tree can be seen on the figure . The objects follow the hierarchy that has been outlined by the domain and the original solution, even with the inconsistencies discussed above 7.1.3.

Note the RAirport class, which is a representation of a real airport. The reason for it to be introduced is that all other classes (instances) are marginal nodes of the hierarchy graph. The airport is the only one that isn't because of the questionable relation stating that a barge is also an airport. It is therefore easier to define new class RAirport for the real instances of an airport and to declare the Airport an interface.

The predicates that are true at the start of the plan, therefore the conditions that have to be contained and true in the initial layer, are parsed into the Model and Object state (MAOstate) class. It consists of several hashmaps, which store the property values according to the string names of the objects. There are no logical checks present, so all properties are considered valid at this point. The MAOstate also contains the daylight TILs and constants, that need to be specified (such as gearbox repair time) or model properties that are constant throughout the model (such as a lease cost of a helicopter).

The Goal predicates, similar to the initial predicates, contain the predicates that must be true at the end, in the final layer. It however doesn't duplicate the information about the constraints and properties of the model.

The links between the places, be it the *link-air* or a *link-land* which serve different purposes, are stored separately in the Link state.

The output of the Parser is a ParsedModel class which contains all the parsed information organized in the aforementioned structures (Initial predicates, Goal predicates, MAOstate and Links) and hashmaps.

7.2.2 The Logical & Filtering layer

The Logical & Filtering layer (L&F) contains some of the more important logic in the model pre-processing, quite resembling the similarities with the *beyond the graphplan* section of the acrticle written by Lopez and Bacchus [27]. Its input is the ParsedModel from the Parser.

The main purpose of the L&F is to construct the classes, implementing the interfaces relevant to the final representation and to decorate those classes with their relevant properties. The L&F is the first layer of consistency checking and if there are some errors in the semantics of the PDDL instance, it is detected here, as it requires all relevant properties to be defined and loaded. Also it prevents the fallacious definitions to have any effect on the model, so even if there is a max windspeed defined for a ship, without the change of the mapper it doesn't have any effect on the model.

The model of the L&F is different from the one in the Parser, although it shares the same basic structure and hierarchy of classes (see figure 7.2). It

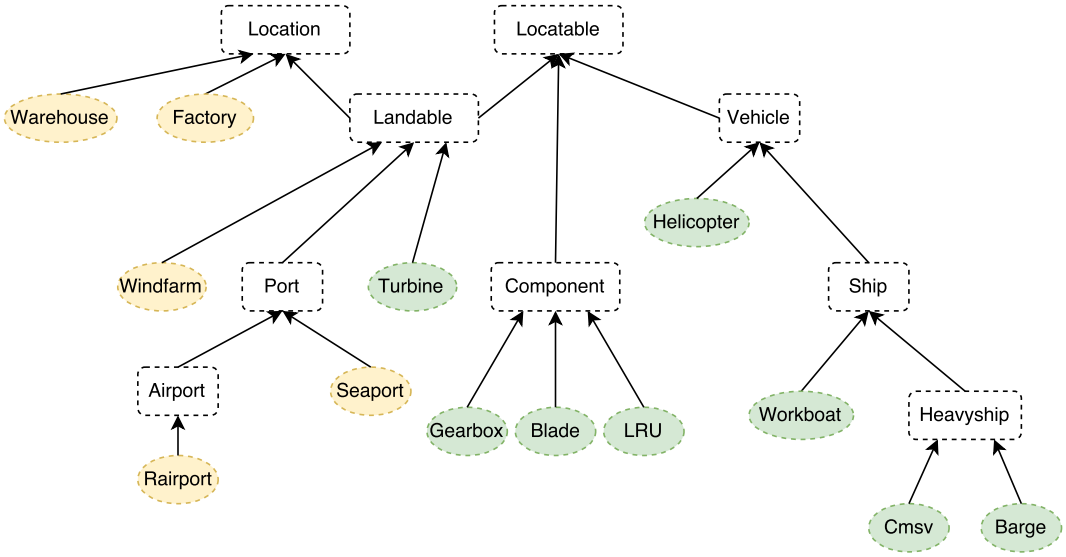


Figure 7.2: The type hierarchy as used in the CSP. The white rounded rectangles are the interfaces. The yellow ovals are classes defined as objects in the Parser and the PDDL, however without any stateful variables and therefore reduced to the collections of IDs. The green ovals represent stateful classes.

is preparing the classes to be of a greater use and through the 'decoration' the important constants and information about the initial and final state are added to the classes. Also, as can be seen on the figure, it is differentiated between the stateful and stateless classes in the CSP model. The stateless classes are de facto constants and do not require to have variables initialized. In the case of mobile vehicles, the information about their possible moves is also recorded and saved in the model.

Another functionality implemented in this layer is a filtering procedure aimed on the removal of irrelevant information in order to make the model and possibilities as small as possible. We can declare and define filters, which statically check different aspects of the model and remove the unreachable values of the model. Details are given below in the subsection 7.3.

One other purpose of the L&F layer is a numerical conversion. Although the authors state that the OPTIC solver operates on the integer domain, the times and lease prices are represented in doubles. It is therefore important to have a unified conversion from doubles to integers. We chose to have the times multiplied by 10 and rounded up and the prices to be multiplied by 100 and rounded.

The output of the L&F is a Model, which is just another collection of information, translated from the ParsedDomain into a more usable form. It is accessed through the CreateModel class and the main translation logic from the PDDL information to the internal representation is done in the Converter

class.

There is one final important thing that happens in the L&F layer - the conversion of the string names to the integer IDs. This part is particularly important for the CP nature of the problem, as the IDs create bases of the domains used in the CP model. Those IDs are created only once and the responsibility of their successful assignment is delegated to the aforementioned Converter class.

7.2.3 The Model

The Model by itself has a responsibility of the planning graph creation and the definition of the CP model along with an algorithm that would lead to the solution of the problem.

The solution search algorithm is a configurable loop, where if there is no solution found in n steps, a graph with $n + 1$ steps is constructed and the search continues. The term *graph construction* encapsulates a set of not only graph construction as it is, but also the creation of the action and global constraints on the model. There is a large amount of types of constraints and actions, see 7.1, and every action requires special instantiation, based on the objects involved. The insides of the plan graph construction are described in the section below 7.4 as it is the main logic part of the model and its design is based on the research we have summarized mainly in section 6.

When a solution is found, the program prints the final state of the graph and its value – variable pairs.

This is the entry point a user will actually use, except for the instance specification file, which determines how will the graph look and what nodes will be generated.

7.3 Filtering in the L&F program layer

There was an idea, implemented in some models [27] that the model should be transformed according to the static knowledge we have before the computation starts. Sometimes those modifications are simple and possibly irrelevant, for example in the work of Lopez and Bacchus [27] the authors propose, that all the variables which have only one value in their scope are therefore constants and should be removed from the model. In our case however we have a great advantage in the form of the domain knowledge. We know exactly what the domain looks like and therefore we may impose more relevant constraints on the model or remove objects and relations that are irrelevant more effectively than the researchers in the general planning case.

This idea gave birth to the notion of filters. Instead of blindly transforming the parsed information into the CP model, we apply the so called filters on the intermediate model. The filters implement an interface accepting the ParsedModel model into its only filter method.

7.3.1 The Vehicle Access Filter

There is a very important filter which modifies the model in a way that it changes its functionality. The helicopters have got a maximum windspeed constraint which makes them unable to travel from/to any location when there is a windspeed higher than allowed for that helicopter. The same applies for ships (and heavyships) with their restriction on the maximum wave height in which they are allowed to travel.

We could keep the information in the actions, so a *fly-to* action could be activated iff the wind speed on both endpoints is lower than the threshold (*max-windspeed*). However, upon the observation of the instance PDDL description we noted that the windspeed and wave height properties are actually constants preset for the instance and do not change over time (as we would expect). This inflexibility of the model allows us to remove all links from the previous ParsedModel leading either from or to the places where this limit is exceeded. The reduction of the number of actions in case of this filter application is significant, as every place is usually connected to all other accessible places in a complete-graph manner. If it is true and we remove one of the n places, we remove $\frac{n*(n-1)}{2} - \frac{(n-1)*(n-2)}{2} = n - 1$ edges. If the initial position is removed, the position still stays in the model, there are just no links to it, so the helicopter/ship can't move and thus participate in the solution.

7.3.2 Other filters

We have got two additional filters, which are order-dependent - the Healthy Turbine Filter and the Healthy Windfarm Filter.

Those filters are based on the relevancy analysis of the locations in relation to the initial and goal state. If the locations or objects are not relevant in any way to the initial state or the final state, they can be removed. This can be very well applied to turbines and windfarms, as the instance description is usually copied with the same number of turbines on the same windfarms, but only a few of them are relevant, because an action (inspection or a repair) is usually required only for a subset of them.

The *Healthy Turbine Filter* therefore removes turbines, which require no action to be executed and don't have any complex conditions in either the initial or the goal state. The filtering however must regard also the prepositions laid on the other objects, because there may be a condition that requires a vehicle to be present at the turbine at the end of the plan and the removal would cause all plans to be unsolvable.

The *Healthy Windfarm Filter* works on a similar principle, removing only the windfarms which have no turbines participating in the solution of the problem and also when *they themselves* don't participate in any initial or final state conditions.

It is possible that more filters could be derived, however with every additional filter it must be taken care not to render the plan insoluble. Other filters, as well as constraints, may also restrain the model, so the solution would be found more quickly, however more complex plans would be impossible to explore.

7.4 The CSP Model – Objects

The CSP model contains the biggest part of the design work in this thesis. It uses the hierarchically organized information from the previous L&F step and extensively uses the Choco Solver technology. [24] The variables used in our model are of an integer nature, from the technology we use IntVars and BoolVars. BoolVars are effectively encoded as Integer Domain Variables (IntVars), however using (Boolean Domain Variables) BoolVars has advantages as special propagation algorithms are implemented just for BoolVars. While exploring the technology, we can refer for example to the *sum* constraint which handles the equality or inequality of a sum of a set of variables to some value. There it is explicitly noted that the BoolVar version of the sum is 'much faster than the one over integer variables'. [24]

As the planning graph we want to construct is quite large, we have to choose an entry point to the model description. Let the first model items described be the objects received from the L&F layer and their mapping into the planning graph. The class for the propositional layers, which we are going to explore first, is called a Fact Layer.

The planning graph of a size n has got $n + 1$ propositional layers and n action layers. The propositional layers contain the variables which describe the state at the time i where $0 \leqslant i < n$ (as for now we consider the plan time to be increasing in an uniform manner as is the case in the classical planning graph). Those variables however have got connections to the states of real items, or 'objects' for that matter. For example if we would have a proposition *at(helicopter, windfarm)*, we can clearly see that in fact it is very relevant to the Helicopter, less to the Windfarm and probably not relevant at all to other objects like Ships. We can therefore organize the propositions (effectively represented by variables) using lists of Java objects.

The IntVar/BoolVar representation of the variables is quite similar to the integer representation of the propositions in the planning graph as proposed in [28]. The propositions that can be effectively true or false (a turbine is activated/deactivated) are very well modelled by the BoolVars. The propositions which have got multiple possibilities (a helicopter can be on a point A, B or C) are modelled by IntVars, where the domain is generated from the IDs of the objects generated in the L&F layer.

It can be noted that the variables of the PDDL objects are in fact strongly connected to the PDDL functions and predicates (albeit not the same). Some

of the PDDL predicates and functions are used as constants because they do not change over time. Therefore they are used in the static filtering in the L&F layer or during the definition of the domains.

Even when we still follow the general hierarchical structure of the model at the transformation of these objects into the CP model is not 1:1 compatible, for example we don't need to have variables for warehouses, ports and windfarms, because they are not stateful and therefore they are omitted.

7.4.0.1 Helicopters

The first model object we translated was a helicopter object. The reason we chose this object to be the first was the testing of the plan, as a helicopter has more constraints than a ship and yet can fulfil some basic tasks and achieve some typical goals present in the problem instances such as inspection or a resetion of a turbine.

A helicopter, or a heli (encoded in the class Heli) has got three stateful variables – the *remaining range*, the*number of engineers aboard* and the position *at* . The domains for the *remaining range* is from 0 to *max-range* defined in the instance. The *number of engineers aboard* is from 0 to the *capacity* defined in the instance. The domain of *at* is a set of integers extracted from the links from the previous model (effectively the ids of windfarms and airports) combined with the ids of the turbines and with a zero added. The meaning of the zero in the domain is the possible position of a helicopter somewhere in between places, for example when it flies or orientates on a windfarm.

7.4.0.2 Turbines

Turbine was the second model to be translated for testing. A turbine is static (it is constantly a part of a windfarm and doesn't move away) yet not stateless. It has got BoolVars *inspected*, *reset* and *operating* with straightforward meanings. It also has got an IntVar which tracks the number of *engineers available* on the turbine. The domain is between 0 and 1, 1 being an arbitrarily picked constant, as there is never a need for more than 1 engineer to be available on the turbine.

7.4.0.3 Workboats, Cmsvs and a Barges

Those three models are all implementing the interface Ship, and Cmsv with Barge also implement the interface Heavyship (these facts are used during the creation of actions). Otherwise they all share the same properties, they all have an IntVar *engineers aboard* with a domain between zero and their capacity and a position with an initialisation similar to the one of the helicopter.

7.4.0.4 Components - Blades, Gearboxes and LRUs

The Blade, Gearbox and LRU objects implement the interface Component. All of them have two IntVars - a variable *part of* with a domain sonsisting of all possible turbine ids and a zero (not part of anything) and a position variable *at*. The domain for *at* is however different for the LRUs, if we accept the inconsistency solving proposal at 7.1.4. In general it contains the IDs of the turbines, barges, cmsvs, seaports and warehouses. LRU however adds the possibilities of helicopters and workboats, as the LRUs are supposed to be small enough to fit on both.

7.4.0.5 Construction of propositional layers

To correctly initialize the propositional layers, based on the information available we create the first one with objects that are defined in the instance. Then we define a cloning procedure for the objects and for the layer and the following layers are created in a same manner.

There are some extra variables in the propositional layer denoting the daylight and its time. The constraints and the logic behind those CP variables are mentioned below The time of the propositional layers is an important factor, as it

7.5 The CSP Model – Action layers and Durative actions

The action layers are collections of durative actions and instantaneous actions. All durative actions consist of three activity BoolVars, *start*, *middle* and *end* (denoted in the program as *s*, *m* and *e*). Additionally they consist of the constraints using these variables to declare effects, conditions and invariants (effetively conditions) of the durative action. The model of the durative actions is heavily inspired by the previous works [31], [32] although the usage is modified.

The durative actions also always contain a constant *duration*, which is defined in the instance or by the model. If it is defined in the model, a constant is used in the program, always converted by the unified convertor from the L&F layer. This 'constant' approach has an advantage, that we eliminate a need of a variable for the actions, however it also has a disadvantage, that we lose a flexibility and therefore the duration-mutable actions like a *fly to* must generate each combination separately.

Firstly, we define that an action is *active*, if it starts, ends or is in progress. The activity is indicated by the *middle* variable being true. Secondly, when an action starts, the *start* variable is true and the same goes for the action end and the *end* variable. Therefore we define valid and invalid combinations

Start	**Middle**	**Stop**	*Effect*
0	0	0	Action is not active
0	0	1	**Invalid**
0	1	0	Action is in progress
0	1	1	Action ends
1	0	0	**Invalid**
1	0	1	**Invalid**
1	1	0	Action starts
1	1	1	Action starts and ends in the next layer

Table 7.2: Valid and invalid combinations of the Durative action BoolVars

of the variable values with the description. Details can be seen on the table 7.2.

The implementation of the restriction of the durative actions is achieved through the *table* constraint with negative examples – we define just the invalid combinations.

7.5.1 Durative actions and time

As was mentioned in the description of the propositional layers, they have got a variable which indicates at which time they start (7.4.0.5). We propose a scheme that would interconnect the activity BoolVars with the integer durations and the propositional layer times. This schemes decorates the durative actions with two additional integer variables named *StartOfDA* and *EndOfDA* with domains from 0 to *MAXTIME* and they signify the start time and the end time of the action respectively. Note that the *MAXTIME* is an arbitrarily chosen constant, chosen to be 10000 in our model.

The *StartOfDA* variable is directly linked to the time of the previous propositional layer on the start of the action. If the action is in progress or ends but doesn't start, then the start value is transferred between the layers by an equality constraint. Mathematically

$$
\begin{aligned}
& l \in [0, size - 1], \\
& \forall a \in ActionLayer_l,\ A = (start_{a,l}, middle_{a,l}, end_{a,l}): \\
& A \in \{(1,1,0),(1,1,1))\} \Rightarrow StartOfDA_{a,l} = Time_{a,l} && (7.1) \\
(l > 0) \wedge & A \in \{(0,1,0),(0,1,1))\} \Rightarrow StartOfDA_{a,l} = StartOfDA_{a,l-1} && (7.2) \\
& && (7.3)
\end{aligned}
$$

where $start_{a,l}$, $middle_{a,l}$, $end_{a,l}$ and $StartOfDA_{a,l}$ belong to the durative action representation a in the current layer, $Time_{a,l}$ belongs to the propositional layer before the durative action and $StartOfDA_{a,l-1}$ is contained in

the representation of the same durative action in the previous action layer. The variable l is constrained between 0 and the size of the graph - 1, although the implication 7.2 is not applicable on the first action layer.

The *EndOfDA* is a bit more complicated. When a durative action ends, it is equal to the time of the following propositional layer. Therefore if an action starts and ends in the same action layer, the time difference between the neighbouring propositional layers must be equal to the duration of the action. The decision whether the action ends or not is not a trivial one if there would be no search direction defined. If however we proceed with the search from the start to end and not in a backward manner, we can define a technique to choose which action should end based on the actions which are active. Logically it has to be the one that has got a smallest endtime defined.

$$l \in [0, size - 1], \forall a \in ActionLayer_l,\ A = (start_{a,l}, middle_{a,l}, end_{a,l}):$$
$$A \in \{(0,0,0)\} \Rightarrow EndOfDA_{a,l} = MAXTIME \tag{7.4}$$
$$A \notin \{(0,0,0)\} \Rightarrow EndOfDA_{a,l} = StartOfDA_{a,l} + duration \tag{7.5}$$
$$end_{a,l} = 1 \Rightarrow EndOfDA_{a,l} = Time_{l+1} \tag{7.6}$$

We can decide which actions end in the current layer when we have decided on the activity of all of the actions. We do it by taking of the minimum out of the *EndOfDA* values. The *MAXVALUE* to which the *EndOfDA* is instantiated when the durative action is not active is a guarantee that when a minimum endtime is picked, it is of the action that currently ends. Note that the condition 7.6 would be enough, however taking the minimum results in better pruning of the model.

$$Time_{l+1} = \min_{\forall a \in ActionLayer_l} EndOfDA_{a,l}$$

The schema can be seen on the figure 7.3.

7.5.2 Durative actions and general sequential constraints

We have defined the valid (or rather invalid) action BoolVar combinations of the durative action, however we also need to define the sequential constraints on actions and their activity variables. We want to ensure that when an action has started, it will be either in progress or finishing in the next layer. Similar reasoning goes for the state when an action has finished or is inactive in the previous action layer – the action can either remain inactive or start.

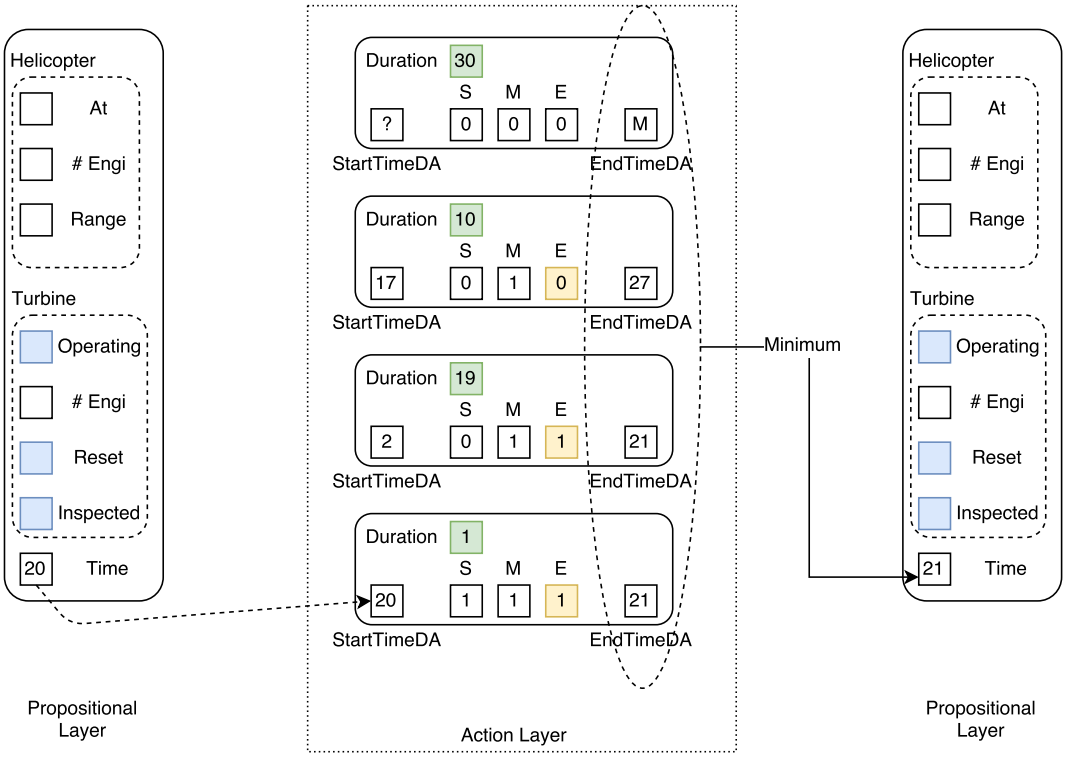


Figure 7.3: The time schema used in the model. Blue variables have a boolean domain. Green cells are constants. Yellow cells show the *end* activity variables of actions which could try to end.

$$
\begin{aligned}
&l \in [1, size - 1] \\
&\forall a \in ActionLayer_l, \\
&A = (start_{a,l}, middle_{a,l}, end_{a,l}), \\
&Group_A = \{(0,0,0), (1,1,0), (1,1,1)\}, \\
&Group_B = \{(0,1,0), (0,1,1)\}: \\
&middle_{a,l-1} = 0 \vee end_{a,l-1} = 1 \Rightarrow A \in Group_A \qquad (7.7) \\
&middle_{a,l-1} = 1 \wedge end_{a,l-1} = 0 \Rightarrow A \in Group_B \qquad (7.8)
\end{aligned}
$$

Here $l \in [1, size-1]$, because we always take into consideration the durative action representation from the previous layer.

7.5.3 Nondurative actions

We have got several nondurative actions in our model, such as embarking and disembarking of the engineers. The advantage of our model is that those

are very easily represented without major modifications to the plan. They are simply considered as actions with their duration equal to 0. To help the model with pruning, we also reduce the possible combinations to two, as the nondurative actions can be either inactive or must start and end at the next layer.

Therefore

$$\begin{aligned} & l \in [0, size - 1] \\ & \forall a \in ActionLayer_l \wedge a \in NondurativeActions \\ & A = (start_{a,l}, middle_{a,l}, end_{a,l}), \\ & A \in \{(0,0,0), (1,1,1)\} \end{aligned} \tag{7.9}$$

7.6 The CSP Model – Durative actions and the PDDL

The PDDL defines properties of the durative actions as the conditions which must hold at the start, at the end or during the whole time the action is executed as well as the start and end effects of the actions. Here we define the combinations of the boolean activity variables of the durative actions and their effects *start*, *middle* and *end*. The combinations are similarly used in [31], however extended by the effects of the start layer.

7.6.0.1 Effects

There are *effects*, that can be present at the start of the action or at the end. Those are bound to the *start* and *end* variables. Here the *Effect* is a set of constraints, usually instantiating values, that must hold in the following propositional layer.

$$\forall a, l \colon start_{a,l} \Rightarrow Effect_{a,l+1} \tag{7.10}$$

$$\forall a, l \colon end_{a,l} \Rightarrow Effect_{a,l+1} \tag{7.11}$$

The equivalent of (at start EFFECT) and (at end EFFECT)

7.6.0.2 Start and End Conditions

The *conditions* of the durative actions are represented in a similar manner as the effects, however they are recorded in the conditions section of the PDDL specification. In the logical model in case of the backward search those conditions are effectively effects that are applied on the previous propositional layer. The *Condition* is a set of constraints that must hold in the starting propositional layer.

$$\forall a, l\colon start_{a,l} \Rightarrow Condition_{a,l} \tag{7.12}$$

$$\forall a, l\colon end_{a,l} \Rightarrow Condition_{a,l} \tag{7.13}$$

The equivalent of (at start CONDITION) and (at end CONDITION)

7.6.0.3 Invariants

The invariants are conditions which must be satisfied throughout the whole execution of the action. Now there is an interesting part - there is a PDDL construct mentioned above, the *over all*. This construct however excludes the endpoints as is defined in the specification [8]. In the source PDDL description the authors do not consider this PDDL property explicitly, so even when logically it would be sound to have the preconditions defined, it is not the case. This *could* cause some troubles if we wouldn't notice it, so instead during the transformation of this condition to our model, we differentiate between the conditions which should hold from the start to the very end and those which should hold in between.

As it may seem, if an action would start and end in the same layer, the *over all* condition would not be enforced at all. If we would bind it only to the *middle* variable, it would be enforced on the start layer as well and that might not be the desired effect. For example when a helicopter is flying and we have a value for the *in flight* state, a condition affecting the start layer as well would make it fail. In general the transformation looks like the following:

$$\forall a, l\colon \neg start_{a_l} \wedge middle_{a,l} \Rightarrow Condition_{a,l} \tag{7.14}$$

The equivalent of (overall CONDITION)

In our model we sometimes omit the invariants, as it is not necessary - there is usually a mutex between the actions of a single vehicle and it can be determined that if an action doesn't change the position of the helicopter and other actions are mutex with that action, the position can't change.

7.7 The CSP Model – Durative actions and the Object types

Every object has its own set of actions. In the program there are action factories for every type or supertype of objects. The supertype factories are useful for action grouping, as some actions are identical for different objects, as can also be seen in the PDDL problem instance specifications (e.g. the Embark-From-Port). It can be noted that all of the actions defined for an object are together limited to one resource – the object itself, therefore we may define a mutex constraint over all of these actions. A simple way to mutex

multiple actions is to create a sum constraint over their *middle* variables and make it lesser or equal to 1 (equation 7.15), therefore either one or zero actions can be active.

$$\forall o \in Objects, l: \sum_{a \in Actions_{o,l}} middle_a \leqslant 1 \tag{7.15}$$

The action descriptions are located between the two layers. It is dependent on instantiation in which state they will be during the search, therefore everything is defined in one set of logical formulas. The actions are connected by preconditions or effect relations with variables from the previous and following (denoted as next) propositional layers. As was noted, 7.4 the variables in propositional layers are organized in objects, usually to which they belong (Vehicle actions to vehicles etc.), but this connection is ambiguous in some cases, where multiple object types are connected to one action. The *Disembark to turbine* action is connected to both a vehicle and a turbine. When we make statements like 'a number of engineers available at a turbine in the following layer is increased by1', effectively we mean that the variable *engineers available* defined in the section 7.4 that corresponds to the aforementioned turbine is increased by 1 relatively to the variable indicating the same information in the previous layer (or there is at least an attempt to increase it by 1, it might fail due to inconsistency). Also, if we write that an action takes a vehicle as a parameter, it effectively means that during the initialization of the action, we require the variables belonging to the vehicle from both the previous and the next propositional layer to be present for us to be able to link them with a logical formula.

For the referencing of the variables, we use function-like notation. To reference a variable *at* of a vehicle in the previous layer, it would be written as $AT_{prev}(vehicle)$. The object variables are written in uppercase, the constants are lowecase and sets and collections start with a capital letter.

7.7.1 Vehicle Actions

The vehicles, in our model represented by ships and helicopters, have got several collective instantaneous actions covering the embarkment and disembarkment of engineers from and to ports and turbines. As all the actions are instantaneous, the logical formulas can be bound to the *start* variable only.

7.7.1.1 Embark from port, Embark bulk from port

These two actions both take a Helicopter as a parameter as well as the list of IDs of the airports and seaports present in the model. The only difference between those actions is that the bulk action loads the vehicle up to its capacity and that the bulk action has a precondition that the vehicle must not be at

full capacity already. Embark form port is represented by a logical formula 7.16 and the bulk version is described by a formula 7.17

$$\begin{aligned} start \Rightarrow & AT_{prev}(vehicle) \in Ports \\ & \wedge ENGIABOARD_{next}(vehicle) - ENGIABOARD_{prev}(vehicle) = 1 \end{aligned} \tag{7.16}$$

$$\begin{aligned} start \Rightarrow & AT_{prev}(vehicle) \in Ports \\ & \wedge ENGIABOARD_{prev}(vehicle) \neq capacity(vehicle) = 1 \\ & \wedge ENGIABOARD_{next}(vehicle) = capacity(vehicle) = 1 \end{aligned} \tag{7.17}$$

7.7.1.2 Disembark bulk to port

Since there is an abundance of engineers, it doesn't make sense to create a special action for a disembarkment of a single engineer. This action takes a vehicle as well as the list of port IDs as a parameter. The vehicle must be at a port and must have some engineers aboard.

$$\begin{aligned} start \Rightarrow & AT_{prev}(vehicle) \in Ports \\ & \wedge ENGIABOARD_{prev}(vehicle) > 0 \\ & \wedge ENGIABOARD_{next}(vehicle) = 0 \end{aligned} \tag{7.18}$$

7.7.1.3 Disembark to turbine, Embark from turbine

Those actions are opposites, one meaning a drop off of an engineer on a turbine and one his pickup. Both take a vehicle and a turbine as parameters. Intuitively, in order for this action to be active, the vehicle must be present at the turbine and it must have some engineers aboard or there must be engineers available at the turbine for disembarkment or embarkment respectively.

Embark from turbine:

$$\begin{aligned} start \Rightarrow & AT_{prev}(vehicle) = id(turbine) \\ & \wedge ENGIABOARD_{prev}(vehicle) > 0 \\ & \wedge ENGIABOARD_{prev}(vehicle) - ENGIABOARD_{next}(vehicle) = 1 \\ & \wedge ENGIAVAIL_{next}(turbine) - ENGIAVAIL_{prev}(turbine) = 1 \end{aligned} \tag{7.19}$$

and Disembark from turbine:

$$\begin{aligned} start \Rightarrow & AT_{prev}(vehicle) = id(turbine) \\ & \wedge ENGIAVAIL_{prev}(turbine) > 0 \\ & \wedge ENGIABOARD_{next}(vehicle) - ENGIABOARD_{prev}(vehicle) = 1 \\ & \wedge ENGIAVAIL_{prev}(turbine) - ENGIAVAIL_{next}(turbine) = 1 \end{aligned} \tag{7.20}$$

7.7.2 Helicopter Actions

The helicopter has a got its own rich set of actions, as there are no other flying vehicles with fuel resources. Helicopters are vehicles, so it contains all the actions from the section 7.7.1. On top of that a helicopter has a set of flying actions to fly between the locations and/or turbines.

7.7.2.1 Fly to

The Fly to action (7.21) has been mentioned a few times already as it is one of the more interesting actions. We define a Fly to action for every air link there is and from the link, which it takes as a parameter, it also loads its duration on initialization. Now, this action has got certain issues regarding the PDDL specification, namely its initial effect $\neg AT(location, helicopter)$. One possibility to which we have the domain prepared is to make an initial effect that would make the helicopter appear at location 0. This was our initial choice, which has its problem - what if the action ends as well? We could also resolve the situation by making sure that no other action involving the helicopter can execute - this is actually quicker, yet more error prone.

$$\begin{aligned} start \Rightarrow & AT_{prev}(heli) = from(link) \\ & \wedge RANGE_{prev}(heli) - RANGE_{next}(heli) = duration \\ & \wedge AT_{next}(heli) = from(link) \\ end \Rightarrow & AT_{next}(heli) = to(link) \end{aligned} \tag{7.21}$$

7.7.2.2 Enter site heli, Leave site heli

The PDDL domain has been designed in such a way, that if we fly from an airport to a turbine, we must first fly to the windfarm and then enter the site (eq. 7.22), therefore fly to a certain turbine. Similarly if the helicopter wants to fly from a turbine to an airport, it must first leave the site and then fly away.

It takes as a parameter the helicopter, an id of the windfarm, a set of turbine ids present at the windfarm and a daylight variable from the previous layer. As it may occur, to successfully navigate the helicopter to a turbine,

there must be daylight. Here is one of the situations where we are not sure about the daylight condition, whether it is necessary to have it at start or at the end. We suppose that it is needed at all times.

$$\begin{aligned} start \Rightarrow & AT_{prev}(heli) = id(windfarm) \\ & \wedge RANGE_{prev}(heli) - RANGE_{next}(heli) = duration \\ middle \Rightarrow & DAYLIGHT_{prev} = 1 \\ end \Rightarrow & AT_{next}(heli) \in TurbinesAtWindfarm \\ & \wedge DAYLIGHT_{next} = 1 \end{aligned} \tag{7.22}$$

A Leave site action doesn't need the daylight at all.

$$\begin{aligned} start \Rightarrow & AT_{next}(heli) \in TurbinesAtWindfarm \\ & \wedge RANGE_{prev}(heli) - RANGE_{next}(heli) = duration \\ end \Rightarrow & AT_{prev}(heli) = id(windfarm) \end{aligned} \tag{7.23}$$

7.7.2.3 Fly between

The Fly between action is an action that allows the helicopter to fly between the turbines of the same windfarm. This action however breaks the assumption that when in flight, the helicopter is at a location 0, because we don't define the Fly between action for every pair of actions, but generally we constrain the helicopter to be on some turbine of the windfarm at the start and on some turbine at the end. This however allows the helicopter to fly from and to the same turbine. This can be resolved by *not* making the helicopter appear at a location 0, but instead to make it remain on the same position until the end of the action and to constrain the from–to relationship . Constraining other actions by mutex (which is done by default) prevents the helicopter from interacting with the turbine, however it appears present at the previous turbine until the very end of the action.

$$\begin{aligned} start \Rightarrow & AT_{next}(heli) \in TurbinesAtWindfarm \\ & \wedge RANGE_{prev}(heli) - RANGE_{next}(heli) = duration \\ middle \Rightarrow & DAYLIGHT_{prev} = 1 \\ end \Rightarrow & AT_{next}(heli) \in TurbinesAtWindfarm \\ & \wedge DAYLIGHT_{next} = 1 \\ & \wedge AT_{prev}(heli) \neq AT_{next}(heli) \end{aligned} \tag{7.24}$$

7.7.3 Turbine actions

Turbine is a stateful object and there are some actions related to it as well. Fortunately it doesn't involve any other objects and therefore they require only turbine objects and some of them the daylight information as well.

7.7.3.1 Enable/Disable turbine

These actions can happen at any time and they are also instantaneous. For the enable turbine action 7.25 it requires that no engineers are on site and that the turbine is not operating. The Disable action 7.26 can be applied whenever the turbine is operating.

$$\begin{aligned} start \Rightarrow & \neg OPERATING_{prev}(turbine) \wedge OPERATING_{next}(turbine) \\ & \wedge ENGIAVAIL_{prev}(turbine) = 0 \end{aligned} \tag{7.25}$$

$$start \Rightarrow OPERATING_{prev}(turbine) \wedge \neg OPERATING_{next}(turbine) \tag{7.26}$$

7.7.3.2 Reset/Inspect turbine

The Reset (eq. 7.27) and Inspect (eq. 7.28) turbine actions are very similar, only the effect and duration differ. They can be executed only during the day and with at least one engineer on site.

$$\begin{aligned} start \Rightarrow & \neg RESET_{prev}(turbine) \\ middle \Rightarrow & \neg OPERATING_{prev}(turbine) \\ & \wedge ENGIAVAIL_{prev}(turbine) = 0 \\ & \wedge DAYLIGHT_{prev} = 1 \\ end \Rightarrow & RESET_{next}(turbine) \\ & \wedge DAYLIGHT_{next} = 1 \end{aligned} \tag{7.27}$$

$$\begin{aligned} start \Rightarrow & \neg INSPECTED_{prev}(turbine) \\ middle \Rightarrow & \neg OPERATING_{prev}(turbine) \\ & \wedge ENGIAVAIL_{prev}(turbine) = 0 \\ & \wedge DAYLIGHT_{prev} = 1 \\ end \Rightarrow & INSPECTED_{next}(turbine) \wedge RESET_{next}(turbine) \\ & \wedge DAYLIGHT_{next} = 1 \end{aligned} \tag{7.28}$$

7.7.4 Ship actions

The ship actions very much resemble the helicopter actions, except for the range constraints as the ships do not have a limited range. The ship actions *Sail to*, *Enter site ship*, *Leave site ship* and *Sail between* correspond to the actions *Fly to*, *Enter site heli*, *Leave site heli* and *Fly between* defined above at 7.7.2. The logical formulae are almost the same.

7.7.4.1 Heavyship actions

The heavyships are ships that are in general slower than the Workboat and also can carry cargo required for the repairs such as wind turbine blades and gearboxes (and LRUs).

7.7.4.2 Load/Unload component

The heavyship actions are therefore a Load component and Unload a component. Both load and unload take a heavyship as a parameter as well as the component.

It is beneficial on the load (eq. 7.29) to make the component appear instantly on the ship, as it prevents other heavyships possibly present at the same location from attempting to load the component as well. The ship that actually loads the component is prevented from further actions until the action finishes.

$$\begin{aligned} start \Rightarrow & AT_{prev}(component) = AT_{prev}(heavyship) \\ \wedge & AT_{next}(component) = id_{heavyship} \end{aligned} \tag{7.29}$$

For the same reason it is more advantageous during the unload (eq. 7.30) to make the component appear at the location at the end of the action, preventing other ships from accessing it before the action finishes.

$$\begin{aligned} start \Rightarrow & AT_{prev}(component) = id_{heavyship} \\ end \Rightarrow & AT_{next}(component) = AT_{prev}(heavyship) \end{aligned} \tag{7.30}$$

7.7.5 Barge actions

The barge has got equipment to replace a wind turbine blade, gearbox or a LRU. Therefore after it approaches the turbine, it can execute the repairs, exchanging the bad part of the turbine for a working one.

7.7.5.1 Replace Blade/Gearbox/LRU

For these operations the barge has got three similar actions – Replace Blade, Replace Gearbox and Replace LRU. The actions have the components as parameters and they look exactly the same for all types of components. It is however necessary to keep them apart as we want only the parts of the same type to be exchanged. The logical formula presented (7.31) represents a general case with a component, nevertheless in the actions the component types are the same.

There are actually two components present in the model as parameters (two pairs), one considered present on a barge and one being part of a turbine at which the barge is currently located. The component on the barge is denoted as *bargecomponent* and the other is called *turbinecomponent*, although after the end of the action the role switches.

$$\begin{aligned} middle \Rightarrow & OPERATING_{prev}(turbine) = 0 \\ & \wedge AT_{prev}(barge) = PARTOF_{prev}(turbinecomponent) \\ & \wedge AT_{prev}(bargecomponent) = id_{barge} \\ end \Rightarrow & PARTOF_{next}(bargecomponent) = PARTOF_{prev}(turbinecomponent) \\ & \wedge PARTOF_{next}(turbinecomponent) = PARTOF_{prev}(bargecomponent) \\ & \wedge AT_{next}(bargecomponent) = AT_{prev}(turbinecomponent) \\ & \wedge AT_{next}(turbinecomponent) = AT_{prev}(bargeecomponent) \end{aligned} \tag{7.31}$$

During the definition of pairs of possible replace actions we may very well control which swaps are possible. In the original PDDL model it has been hardcoded in the instance which component should end where.

7.7.6 Component actions (Land operations)

Even though the model mostly revolves around the vehicles and the actions on the sea, we need an action that would connect the warehouses to the ports to allow the transportation of the components. The links are declared in the instance usually in a complete graph manner. If it wasn't true, it is easy to create the complete graph if it is connected.

7.7.6.1 Deliver component

The deliver component takes as a parameter the component and a land link. It is a simple action that moves the component between two land-linked locations.

$$\begin{aligned} middle \Rightarrow & AT_{prev}(component) = from(link) \\ end \Rightarrow & AT_{next}(component) = to(link) \end{aligned} \tag{7.32}$$

7.7.6.2 Timed initial literals – Daylight

Although it doesn't correspond to any object, the Daylight TILs also fall into the category of actions in our model. It is very easily represented by actions, which have predefined durations.

7.7.6.3 Day and Night

The Daylight TIL is represented by two actions. The Day action (eq. 7.33) has a duration of 16 hours (appropriately transformed) and the Night action (eq. 7.34) has a duration of 8 hours. Using the sequential constraints we will connect them together so when one action ends, the other must start in the following layer. 7.8.3 The effects of the actions are straightforward – they determine the state of the daylight variables which do not belong to any object. Therefore instead of having objects on its input, the actions are using the BoolVars of the previous and following propositional layers.

$$\begin{aligned} middle &\Rightarrow daylight_{prev} = 1 \\ end &\Rightarrow daylight_{next} = 0 \end{aligned} \tag{7.33}$$

$$\begin{aligned} middle &\Rightarrow daylight_{prev} = 0 \\ end &\Rightarrow daylight_{next} = 1 \end{aligned} \tag{7.34}$$

Note that we don't have to define the frame axiom for the daylight, as the actions determine the values by themselves. In the program the frame axiom constraints are redundantly created even for this variable.

7.8 The CSP Model – other constraints and the frame axiom

Now we have defined the scheme for the creating of the planning graph, we have created the durative actions and constrained their effects, preconditions and possible states they can appear in.

7.8.1 Initial and Final state

Every problem instance has its initial state and some goals to be achieved. To encode this information into the model, we need to constrain the values of the variables in the first and last propositional layer. In our program it is done through the InitialConditionFactory and the FinalConditionFactory. The states are defined again object-wise. In theory some variables do not have to have a defined state on the start because we don't know/care about the state at the time. In our model almost all initial state specifications are

required. It doesn't apply to the goal state though, so the final position of the components can be left blank as well as the operation of turbines. For the vehicles however it is necessary to have them in some state at the end, because we rent them and want to have them returned to the lessor.

All the constraints are simple arithmetic constraints determining the initial and final value.

The model is prepared for the graph extension – the goal constraints are generated and collected together and then posted. The reference is retained so they can be unposted, the graph extended and new goal constraints generated. It is not fully implemented in the program as for now, so with every planning graph size increase the old graph is discarded and replaced by a newly constructed one.

7.8.2 The Frame axiom

Throughout the design we have mentioned the frame axiom a few times. We use the implementation inspired by [34], therefore we record the affecting actions for every variable and we make the model observe whether they are activated. There are alternative approaches where the actions are recorded as achievers of a positive or negative value of a variable. When those actions are active, the corresponding value is set. When no achiever is active, the value is the same as it was in the previous layer.

In our model we have got two catches. The first catch is that we are operating with the variables which have integer domains. If we were to literally copy the approach, we would have to record the achievers of the *values* in those domains. Therefore for every variable we record an action that has the variable in its effects.

The second catch is with the durative action design. Normally we would collect the activity variables of the actions and decide on the value based on them being true or false. The durative actions however have possibilities of effects when they start, end or are in progress. We have designed a scheme, where on creation of the durative actions we can record a specific activity BoolVar of the action and a variable it affects. For example for the Unload action at 7.30 it may be seen that the *at* variable of a component is modified at the end of the action, therefore we record the pair $(end, AT_{next}(component))$.

We collect the supports into a (hash)map *Support* with lists as values and the variables as the keys and then create a constraint for every variable.

$$\forall v_l \in PropositionalLayer_l \colon \Big(\sum_{\forall s \in Support(v_l)} s = 0 \Big) \Rightarrow v_l = v_{l-1} \tag{7.35}$$

7.8.3 Global constraints and Sequential constraints

To help the model with solving a problem, it is good to add some extra constraints to help it prune the search tree. One of the useful constraints is a constraint that limits the number of action usages to 1 or to force a condition that in every action layer at least one action must finish. This limits the model to a subset of the possible solutions, so the search isn't complete.

7.8.3.1 Helicopter flight

The helicopter has one interesting property, which has to be considered during modelling – it flies and all the time it flies it consumes the fuel, lowering its remaining travelling range. We create a sequential constraint on the Fly to actions, where the *to* part of the action is a windfarm, to enforce the Enter site heli action. The helicopter can land on a turbine to preserve fuel and also it wouldn't make sense to fly to a windfarm and then to wait until a certain time.

The constraint is formed in the following manner:

$$\begin{aligned}
&\forall wto \in Windfarms,\\
&\forall flyto_l \in FlyTo(heli, from, to = wto)_l,\\
&\forall enterSite_{l+1} \in EnterSite(heli, windfarm = wto)_{l+1}:\\
&\Big(\sum_{\forall flyto_l} end(flyto_l)\Big) > 0 \Rightarrow start(enterSite_{l+1})
\end{aligned} \tag{7.36}$$

The same goes for the Leave site action. After the helicopter leaves the site, we want it not to hover and wait, but to fly somewhere straight away because otherwise we could always find a plan better than the one where it is allowed to hover.

$$\begin{aligned}
&\forall wfrom \in Windfarms,\\
&\forall flyto_{l+1} \in FlyTo(heli, from = wfrom, to)_{l+1},\\
&\forall leaveSite_l \in LeaveSite(heli, windfarm = wfrom)_l:\\
&start(leaveSite_l) \Rightarrow \Big(\sum_{\forall flyto_{l+1}} start(flyto_{l+1})\Big) = 1
\end{aligned} \tag{7.37}$$

7.8.3.2 Fly force not fly

Non-mandatory condition can be also applied on the fly action. As there are always routes defined for the complete graph (or the graph can be easily transformed to such form), we can constrain the flight action not to happen after the flight. The effectiveness is questionable, because some solvers reported

```
TwofoldImplication(BoolVar activity, Constraint constraint):
// standard A => B
model.ifThen(activity, constraint)

// logically equivalent notB => notA
model.ifThen(not constraint, not activity)
```

Figure 7.4: Twofold constraint posting for implications

that additional sequence constraints actually *worsen* the performance of the model. [27]

$$\begin{aligned}
&\forall flyto_l \in FlyTo(heli, from, to)_l, \\
&\forall flyto_{l+1} \in FlyTo(heli, from, to)_{l+1}: \\
&\sum_{\forall flyto_l} end(flyto_l) > 0 \Rightarrow \sum_{\forall flyto_{l+1}} start(flyto_{l+1}) = 0
\end{aligned} \tag{7.38}$$

7.8.4 Propagation enhancement

During the experimentation with the model we have noticed, that the implications prune the model in a one-sided way. Through the logical equivalency we know that $A \Rightarrow B = \neg B \Rightarrow \neg A$. The Choco solver however doesn't recognise this equivalency and as we have used the implications extensively throughout the definition of the preconditions and effects, it is beneficial for the pruning strength of the model to post even the transformed constraint. What it means is that if we have two BoolVars A and B and a constraint $A \Rightarrow B$ and $B = 0$, after propagation the domains would still be $Dom(A) = \{0, 1\}$ and $Dom(B) = \{0\}$, although the value 1 is not acceptable any more.

Therefore we have defined a method, that posts both forms of the implication, taking a BoolVar and a constraint as parameters to match the most frequent use in the model.

7.8.5 Cost optimization

There were concerns about the price optimization in the original solution, as the solver considered the total price *after* finding a solution, therefore too late to utilize it during the search. We propose a model modification and a specification to help the model find a solution. We also analyze the setbacks mentioned in the previous solution and whether our model tackles them.

7.8.5.1 Cost computation

The cost computation proposed is based on a simple idea – the vehicle must be rented if it doesn't appear on its starting/finishing position (they are usually the same). If it is indeed rented, there is an IntVar variable for each vehicle, which is then set to the time of the previous propositional layer subtracted from the time of the current propositional layer. The total lease cost of a vehicle is therefore determined by a sum of the values of those variables. The total lease cost, the target optimization variable, is computed as a sum of the lease costs of the vehicles.

$$\forall v \in Vehicle:$$
$$leaseCost(v) = leasePerHour(v) * \sum_{\forall l:\, atHome(v)_l} Time_l - Time_{l-1}$$

$$totalCost = \sum_{\forall v \in Vehicle} leaseCost(v) \tag{7.39}$$

The great advantage is that using a good propagation algorithm for a sum, after finding an initial solution, the other solutions are more effectively pruned. The model can use the domain filtering to make a node in the search tree inconsistent when the total time so far is greater than the previously found solution.

7.8.6 Possible issues

One of the first issues of the cost computation is the definition itself. Perhaps it would be necessary to join the lease into one big lease action which must start as late as possible and finish as soon as possible. This is a bit unrealistic in the sense that we can rent the vehicles at different days, not paying for the time between. There could be constraints defined for theduration of the time the vehicle is not leased, but those would be tricky to implement.

Another issue, not so problematic, is to enforce the leased state when there is a component located on the heavyshiip. The exact conditions would have to be discussed however, because usually the model doesn't include the position of faulty components in the goal state, so even in the PDDL model it may remain on the ship in the end.

7.9 Search

The extensive modelling work we have done has been done at the expense of the CSP search research. The theory behind the search was to simulate a

chain-forward procedure, as was the case in the base solution 3.3. The chain-forward procedure would require a hard-coded order of the decision variables as well as deciding which of the variables in the model are indeed needed for the decision.

The direction of the search is advantageous for several reasons. First, while instantiating the n-th layer, we have the previous $n-1$ layers already fixed and are sure that the state the layer corresponds to is reachable from the initial state. Secondly, we are working with time and resources, which are depleting over time. As we are sure, that the current state is reachable from the initial state, we also have current values for the time and resources. If we would use a backward search as in the usual planning graph models, which do not have durative actions, we wouldn't have much information about the time or the resources until the first layer was reached.

The planning graph models usually use the variables indicating the activity of the actions as decision variables. The same goes for our model, we use the action activity variables as decision variables. Since there are three activity variables per action and our primary direction of the search is start to goal, it is reasonable to instantiate the variables by layers. In every layer we instantiate the *start* activity variables first, *middle* second and *end* last.

This approach however wasn't very thoughtful, as there were simply too many possibilities to explore and the search got very slow for greater instances. We cover the failure of the search in the details.

Because of the failure of the direct forward-chaining-like approach, we experimented with some other searches present in the model. The backward search approach resulted in an even worse time than the forward approach. Finally we settled for the activity based search, which has proved to be the best for the current model.

Chapter 8

Results

8.1 Experiment

We have used just the simple minimal domain with one or two vehicles and we have observed the behavior of the model. While testing the searches, there was no need to set up a statistical evidence of the behaviour and to statistically prove the significancy of the results, because usually the directional search as proposed in 7.9 took hours to complete or didn't complete at all for more complex instances, while the Activity-Based search found a solution in minutes.

The experiment has been done on a computer with 16GB of RAM memory and an Intel Core i3-6100 processor.

8.2 Modelling

We have created a model which is flexible enough to contain all the features required by the problem. The framework is structured so it is easy to add new actions after their definition. Regarding the expressivity and completeness of the model, it is a success, as it tackles some more issues present in the domain, including the use of the previously found total cost as a pruning tool during the search. Also we have identified some under-specification and inconsistencies in the domain, which could be used in its redefinition or update.

8.3 Search results and implications

While trying to find solutions, we ran into very big issues with its computational complexity. As the durative actions model has 3 activity BoolVars instead of 1, the number of possibilities has risen significantly. With the number of possibilities also the time rises, so the plan length we can check is prohibitively limited. For example it is acceptable to make a plan for one

helicopter and one turbine inspection. The time it takes however is $\approx 23s$ if we know the number of layers it will require. If we need more inspections, the time needed to solve the model rises even more.

A similar complexity rise we see on a vehicle addition. A vehicle alone consists of 2-3 variables in every layer, which is not a huge raise. The actions and the constraints generated in every of the $n-1$ action layers is very high nonetheless, and the combination of possible starts and ends is extremely high. For example for every solution we find for treatment of an inspection requirement of a turbine by a ship, by adding another ship we multiply the number solutions by 2^k while taking into consideration only the actions where the unused ship does nothing but embark and disembark engineers. This is a huge setback of the model we chose, which encodes everything at once.

Another reason for the slowness is that we don't know the number of steps required for the plan. Even if it would be short enough to finish in a time of days, it doesn't have a fast fail mechanism, which would quickly scan the problem and find that it is unsolvable. Therefore it takes a lot of time exploring smaller plans, which have no solution. Because during the search we don't know that the plan of a lesser than acceptable length doesn't have a solution, we have to perform a full search which would prove that it indeed doesn't.

8.3.1 Example

For example we can take an instance of a problem, whose optimal graph length is 9, it has only one inspection to do but it can choose from two vehicles – the helicopter and a workboat. A plan with only one choice of the vehicle finds a plan in less than 2 minutes or 23s if we correctly estimate the plan length in advance. When the choice of the two vehicles is introduced, the search takes 6 hours or 3.2 hours if we know the plan length in advance. For the vehicle choices we present a graph monitoring the rising time complexity and the number of variables and constraints present (figure 8.1).

We can see that, since the y-axis is logarithmical, the time complexity raises exponentially.

8.3.2 Possible resolution of the issues

8.3.2.1 Decomposition

As we can see, the all-in-one planning is inapplicable. We could improve the problem through decomposition, as we proposed at 7. The plans can be decomposed by vehicles and problems, effectively shrinking the problem to acceptable sizes. It would still not be enough for all the issues, as even a problem with one helicopter and multiple turbines takes a lot of time to finish. The question we would have to deal with is a quick rough estimation

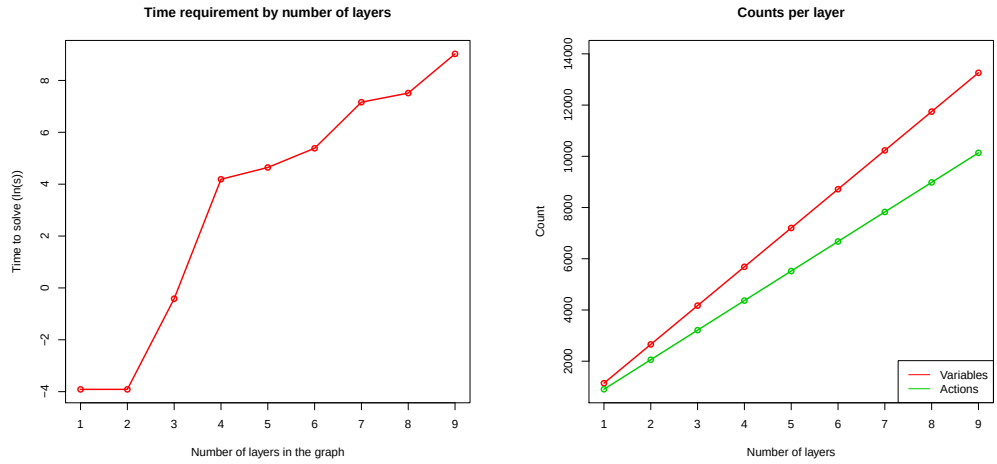


Figure 8.1: The time and the number of constraints and variables by the number of layers

of the solution efficiency, so we could separate the goals and pair them with the vehicles assigned to those goals.

8.3.2.2 Omission and reformulation

There have been many improvements on the model through the static information at hand. There are many more possibilities to precompute and select many properties of the model. For example we can be sure that renting a barge or use it for simple reset & inspection work is probably never a solution, because the cost of the rental of a barge for an hour is ≈ 17 times more than a rental of a helicopter and ≈ 25 times more than the rental of a workboat. Similar, yet lesser counts go for the cmsvs. Based on this knowledge, we might figure out that we will *never* use a cmsv or a barge for the simple work, therefore the reset & inspection tasks should be divided between the workboat and helicopter only.

The same goes for the barge. As there is actually no restriction on the capacity, the barge can load an unlimited number of components onboard. As it doesn't have fuel requirements, the problem is completely reduced on the route finding. If there was a limitation of only one item onboard, then there would be the interesting role played by the cmsv, which can reload the turbine components onto the barge and transport them from the ports while the barge does the maintenance on the turbines.

Even the barge scenario has its *buts* however. As we have noticed in the section 7.1.4 and after some research in the industrial domain, assigning a barge to a replacement of a LRU is a resource-wise overkill, since the LRUs are supposed to be easily replacable units. It is possible that only the barge

has got the equipment to make such a replacement. We are quite sceptical about that claim and it would require more consultations with the industrial specialists which helped in the creation of the problem domain.

Another twist in the barge scenario is the scheduling part for the cmsv in case of the capacity constraints (defined after consultation with industrial specialist). As every repair takes a lot of time, in all the domains it requires 184 hours to replace a blade or a gearbox, the scheduling for the cmsv is trivial. Every time the barge would be about to finish an operation, the cmsv would be rented, the next component loaded and both of the heavyships would arrive at the point of the next repair at the same optimal soonest possible time. Then the component would be loaded onto the barge, the repair would start and the cmsv would return into its base port and its lease would end for another week.

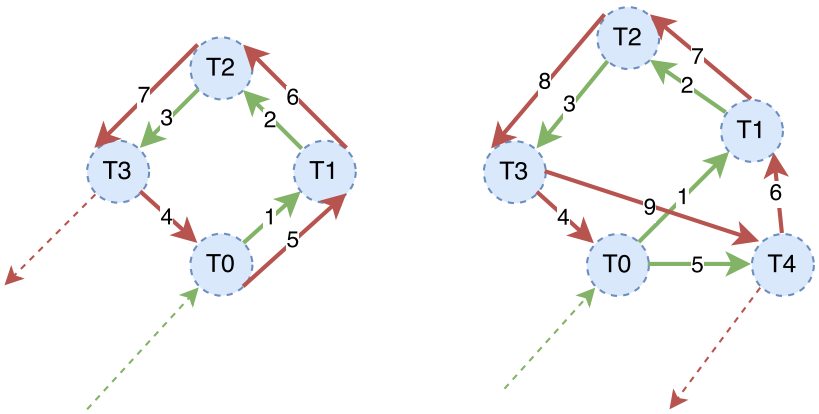
8.4 Further pointers

The information in 8.3.2.2 lead us to an easier an much more lightweight representation, which would surely overshadow our model in performance while losing flexibility. The alternative model we could use would only work with subproblems, which could, by themselves, be further reduced.

Take for example the structure of the windfarms and the turbines located at the windfarms. Although a good organizational structure to maintain, the information of a windfarm can be stripped in the CSP model, because we don't need it in any way. If there is some limitation present, be it the windspeed or the wave height, we know it in advance thanks to the staticity of the parameter in the domain and therefore we can apply the filters as we have done in our case. While stripped of the windfarm information, we can define straight links between the turbines and ports and work with them.

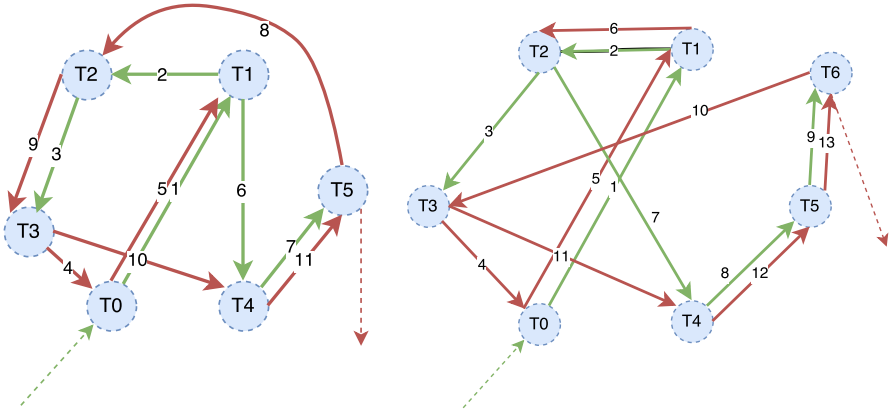
To help the computation even more, we might try to use the information about the inspection and reset duration, so we could answer a question – how many engineers does a vehicle need to service all the turbines in the windfarm while minimizing the time and therefore the cost? Now we are talking about the scheme where the vehicle would leave an engineer on the turbine and fly/sail to other turbine, returning for the engineer eventually.

The answer to the question above is – it depends. If only resets are needed, the answer is 1, because the reset take the same time as the travel between the turbines. We choose either to wait for the reset to finish and to embark the engineer right afterwards, doing only one visit to the turbine which makes up to one hour per turbine (30 mins for the reset, 30 min transport to the next turbine), or (if we had unlimited number of engineers aboard) we would only drop them off and travel around, doing the work and travel in parallel. This would however mean that we would have to visit every turbine twice, so while we would do some work in parallel, in the end it again adds up to 1 hour



(a) a solution for 4 turbines

(b) a solution for 5 turbines



(c) a solution for 6 turbines

(d) a solution for 7 turbines

Figure 8.2: Optimal solutions for serial inspection with a vehicle using 4 engineers

(30 minutes per travel to the turbine, 2 travels needed). This is regardless of the vehicle, because the *time distance* between the turbines is the same for helicopters and ships.

The answer for the inspections on the same windfarm is 4. We suppose that there is a lot of turbines to be inspected and again we can do the reasoning the same way as for the resets. We either have to wait, which adds up to 2.5 hours (30 mins of transport, 2 hours of inspection), or we can pick up the engineer when his work is done, therefore visiting every turbine twice. Since the inspection time takes 2 hours, the number of steps after which we can return is 4. Another resoning is that since we have to visit every turbine twice and we start on a random turbine of a windfarm, the optimal number of steps is $n * 2 - 1$ and we can easily propose the solution that covers that case (fig. 8.2).

For higher numbers of turbines, the schema is similar. For n turbines on the same windfarm, we decompose the turbines to $n = x * 4 + a$, where $a \in [4, 7]$. For a turbines the schema from the figure 8.2 apply. For $x * 4$ turbines the solution for 4 turbines apply and they are interconnected by the initial/final edges denoted by the red and green arrows on the schema.

The optimal number of engineers to solve the problem with inspections is important in relation to the capacity of the vehicles. The workboat, ≈ 3.3 times slower than the helicopter yet ≈ 1.5 times less costly has a maximal capacity of 5 engineers. Therefore we know that when it arrives on the windfarm, it may service the turbines nonstop, maybe until a daylight constraint is reached. A question remains – when is it more advantageous to use the workboat rather than a faster heli? We can compute an estimate for every windfarm as was done in the equation 8.1 and therefore the least number of turbines required for the workboat to be more efficient.

$$
\begin{aligned}
&helirange = 8.5, \\
&timeToTravel_{heli} = 1.7, \\
&timeToTravel_{workboat} = 5.5 \\
&\frac{costHeli}{costWorkboat} = 1.5, \\
&refuelTime_{heli} = 0.5:
\end{aligned}
$$

$$
\begin{aligned}
&1.5 * \frac{t}{8.5 - 1.7 * 2} * (0.5 + 2 * 1.7) + 1.5 * t < 11 + t \\
&t < 6.7
\end{aligned} \tag{8.1}
$$

This only addresses a very specific problem present only on one windfarm, the situation would be different if we took into consideration other windfarms as well, however it well demonstrates the hidden simplificating property of the problem.

Conclusion

We have successfully implemented a model that contains all the information encoded in the source PDDL domain. It works well for small instances of the problem, nevertheless for bigger instances or for more vehicles, as required by the source instances, there would have to be further research into the search algorithm of the CSP.

We have explored implementation possibilities and technologies around the planning graph approach to our problem, which is at the border of scheduling and planning, taking the complexities of both. We have designed a constraint model that extends the traditional planning graph that utilizes time and timed initial literals, which has not been done in any of the articles researched. In the process of the implementation we took inspiration from various papers using different technologies to deal with either durative actions in the CSP or improving the representation of the planning graph.

Finally we have analyzed the issues present in the model and have proposed pointers for the implementation of a different CSP model (or another search algorithm in general, as the information we analyzed is universal), which would use the decomposition and simplification of the problem. Through the simplification and utilization of the outlined properties the computational model created using the information analyzed would be much simpler and faster, although lacking the flexibility of the model implemented by us.

Bibliography

[1] Pattison, D.; Xie, W.; et al. The WINDY domain ? a challenging real-world application of integrated planning and scheduling. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, edited by D. Borrajo; S. Kambhampati; A. Oddi; S. Fratini, Association for the Advancement of Artificial Intelligence (AAAI), 2013. Available from: http://strathprints.strath.ac.uk/47365/

[2] Barták, R. Guide to constraint programming - Consistency Techniques. 1998, http://ktiml.mff.cuni.cz/~bartak/constraints/consistent.html.

[3] Mohr, R.; Henderson, T. C. Arc and Path Consistence Revisited. *Artif. Intell.*, volume 28, no. 2, Mar. 1986: pp. 225–233, ISSN 0004-3702, doi:10.1016/0004-3702(86)90083-4. Available from: http://dx.doi.org/10.1016/0004-3702(86)90083-4

[4] International Renewable Energy Agency. Wind Power Technology Brief'. March 2016, http://www.irena.org/DocumentDownloads/Publications/IRENA-ETSAP_Tech_Brief_Wind_Power_E07.pdf.

[5] International Renewable Energy Agency. Renewable Energy Statistics 2016'. July 2016, http://www.irena.org/DocumentDownloads/Publications/IRENA_RE_Statistics_2016.pdf.

[6] Ghallab, M.; Howe, A.; et al. PDDL - The Planning Domain Definition Language. October 1998, http://icaps-conference.org/ipc2008/deterministic/data/mcdermott-et-al-tr-1998.pdf.

[7] Edelkamp, S.; Hoffmann, J. PDDL2.2: the Language for the Classical Part of the 4th International Planning Competition. 2004, http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/Resources/edelkamp-hoffmann-tr-2004.pdf.

[8] Fox, M.; Long, D. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Int. Res.*, volume 20, no. 1, Dec. 2003: pp. 61–124, ISSN 1076-9757. Available from: http://dl.acm.org/citation.cfm?id=1622452.1622454

[9] Blum, A. L.; Furst, M. L. Fast planning through planning graph analysis. *Artificial intelligence*, volume 90, no. 1, 1997: pp. 281–300.

[10] Benton, J.; Coles, A. J.; et al. Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *ICAPS*, volume 77, 2012, p. 78.

[11] Coles, A.; Coles, A.; et al. Forward-chaining Partial-order Planning. In *Proceedings of the Twentieth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'10, AAAI Press, 2010, pp. 42–49. Available from: http://dl.acm.org/citation.cfm?id=3037334.3037341

[12] Rossi, F.; Van Beek, P.; et al. *Handbook of constraint programming*. Elsevier, 2006.

[13] Ginsberg, M. L.; Frank, M.; et al. Search Lessons Learned from Crossword Puzzles. In *AAAI*, volume 90, 1990, pp. 210–215.

[14] Wallace, R. J. Why AC-3 is Almost Always Better Than AC-4 for Establishing Arc Consistency in CSPs. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 239–245. Available from: http://dl.acm.org/citation.cfm?id=1624025.1624059

[15] Bessière, C. Arc-consistency and arc-consistency again. *Artificial Intelligence*, volume 65, no. 1, 1994: pp. 179 – 190, ISSN 0004-3702, doi: http://dx.doi.org/10.1016/0004-3702(94)90041-8.

[16] Bessière, C.; Régin, J.-C.; et al. An Optimal Coarse-grained Arc Consistency Algorithm. *Artif. Intell.*, volume 165, no. 2, July 2005: pp. 165–185, ISSN 0004-3702, doi:10.1016/j.artint.2005.02.004. Available from: http://dx.doi.org/10.1016/j.artint.2005.02.004

[17] Chmeiss, A.; Jégou, P. Path-consistency: When Space Misses Time. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, AAAI Press, 1996, ISBN 0-262-51091-X, pp. 196–201. Available from: http://dl.acm.org/citation.cfm?id=1892875.1892904

[18] van Hoeve, W.-J. The alldifferent constraint: A survey. *arXiv preprint cs/0105015*, 2001.

[19] Barták, R. Lecture notes Programování s omezujícími podmínkami. 1998, http://ktiml.mff.cuni.cz/~bartak/podminky/lectures/lecture08.pdf.

[20] Laburhe, F. CHOCO: implementing a CP kernel. In *Techniques for implementing constraint programming Systems, a post-conference workshop of CP-2000*, edited by N. Beldiceanu; W. Harvey; M. Henz; F. Laburhe; E. Monfroy; T. Muller; L. Perron; C. Schulte, Science Drive 2, Singapore 117599: Morgan Kaufmann Publishers Inc., 2007, pp. 71–85, 55. Available from: http://cse.unl.edu/~choueiry/Documents/Choco.pdf

[21] Downing, N.; Feydy, T.; et al. Explaining Alldifferent. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, ACSC '12, Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2012, ISBN 978-1-921770-03-6, pp. 115–124. Available from: http://dl.acm.org/citation.cfm?id=2483654.2483668

[22] Gent, I. P.; Miguel, I.; et al. Generalised arc consistency for the AllDifferent constraint: An empirical survey. *Artificial Intelligence*, volume 172, no. 18, 2008: pp. 1973 – 2000, ISSN 0004-3702, doi: http://dx.doi.org/10.1016/j.artint.2008.10.006. Available from: http://www.sciencedirect.com/science/article/pii/S0004370208001410

[23] OscaR Team. OscaR: Scala in OR. 2012, available from https://bitbucket.org/oscarlib/oscar.

[24] Prud'homme, C.; Fages, J.-G.; et al. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. Available from: http://www.choco-solver.org

[25] van Beek, P.; Chen, X. CPlan: A Constraint Programming Approach to Planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, Menlo Park, CA, USA: American Association for Artificial Intelligence, 1999, ISBN 0-262-51106-1, pp. 585–590. Available from: http://dl.acm.org/citation.cfm?id=315149.315406

[26] Do, M.; Kambhampati, S. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, volume 132, no. 2, 11 2001: pp. 151–182, ISSN 0004-3702, doi:10.1016/S0004-3702(01)00128-X.

[27] Lopez, A.; Bacchus, F. Generalizing graphplan by formulating planning as a CSP. In *IJCAI*, volume 3, 2003, pp. 954–960.

[28] Barták, R.; Toropila, D. Reformulating Constraint Models for Classical Planning. In *FLAIRS Conference*, 2008, pp. 525–530.

[29] Barták, R. A novel constraint model for parallel planning. *PlanSIG2010*, 2010: p. 15.

[30] Barták, R.; Toropila, D. Enhancing Constraint Models for Planning Problems. In *FLAIRS Conference*, 2009.

[31] Barták, R. A Flexible Constraint Model for Validating Plans with Durative Actions. *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice. Frontiers in Artificial Intelligence and Applications*, volume 117, 2005: pp. 39–48.

[32] Fox, M.; Long, D. Fast Temporal Planning in a Graphplan Framework. In *AIPS Workshop on Planning for Temporal Domains*, volume 2, Citeseer, 2002, pp. 9–17.

[33] Michel, L.; Van Hentenryck, P. Activity-based search for black-box constraint programming solvers. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, Springer, 2012, pp. 228–243.

[34] Barták, R.; Salido, M. A.; et al. New trends in constraint satisfaction, planning, and scheduling: a survey. *The Knowledge Engineering Review*, volume 25, no. 03, 2010: pp. 249–279.

Appendix A

Acronyms

AC Arc consistency

CP Constraint programming

CSP Constraint satisfaction problem

COP Constraint optimization problem

FC Forward checking

GAC General arc consistency

GCC Global cardinality constraint

PDDL Planning domain definition language

RC Required concurrency

TIL Timed initial literals

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `ChocoProject` ..... the directory of the main maven project
- `PreprocessingAndModel` ..... the directory of the required logical transformation maven project
- `PDDLParser` .... the directory of the required PDDL parser maven project
- `domains` ... the directory with the PDDL domains of the original problem
  - `plans` ..... the directory with re-created plans by the OPTIC solver
  - `problems` ..... the directory of the original PDDL problem definitions
  - `newproblems` ... the directory of the testing PDDL problem definitions
  - `domain-windy-complex.pddl` .. the modelled complex PDDL domain
  - `domain-windy-simple.pddl` ..... the simplified PDDL domain
- `windy` ..... the directory with the Windy domain information
- `thesis` ..... the directory with the thesis source code
- `DP_Prochazka_Martin_2017.pdf` ..... the thesis in the pdf format